

# Transformation Digitale Accélérée : Repenser les Déploiements Cloud à travers Kubernetes pour Favoriser l'Innovation

Mohcine YAHIA

M1 APP RS 2021-2024

Consultant Cloud

Maître d'apprentissage : **Arnaud TORRES**

Tuteur Enseignant : **Karim BAZIZI**

## Sommaire

<b>Summary .....</b>	<b>3</b>
<b>Introduction .....</b>	<b>4</b>
<b>I. Contexte.....</b>	<b>5</b>
1.1) Pipelines CI/CD : Concepts et définition.....	5
1.2) Le déploiement applicatif.....	6
1.2.1) <i>En général</i> .....	6
1.2.2) <i>Chez Fujitsu</i> .....	7
<b>II. Projet : Déploiement d'un cluster Kubernetes pour la CI/CD.....</b>	<b>9</b>
2.1) Présentation du Projet .....	9
2.2) Architecture cible .....	10
2.2.1) <i>Azure DevOps : Vue d'ensemble</i> .....	11
2.2.2) <i>L'infrastructure requise</i> .....	13
2.2.3) <i>Interconnexion avec les projets applicatif</i> .....	16
2.3) Déploiement de l'infrastructure.....	17
2.3.1) <i>Rédaction de l'Infrastructure-as-Code</i> .....	18
2.3.2) <i>Création des pipelines CI/CD</i> .....	20
2.4) Construction des images Docker .....	23
2.4.1) <i>Déploiement d'un Self-Hosted-Agent</i> .....	24
2.4.2) <i>Personnalisation des images</i> .....	26
2.5) Déploiement dans Kubernetes.....	29
2.5.1) <i>Prérequis</i> .....	29
2.5.2) <i>Déploiement des Pods</i> .....	30
2.5.3) <i>Mise à l'échelle avec KEDA</i> .....	32
2.5.4) <i>Création de Helm Charts</i> .....	34
<b>III. Bilan .....</b>	<b>36</b>
3.1) Résultats obtenus .....	36
3.2) Analyse et avis sur le sujet .....	37
3.3) Apport personnel et apprentissage.....	38
<b>Annexes.....</b>	<b>39</b>
<b>Table des références.....</b>	<b>42</b>
<b>Table des figures .....</b>	<b>43</b>

## Summary

This thesis focuses on improving the deployment process of the AMCS team at Fujitsu France. In a DevOps context, the project aims to deploy a private AKS cluster capable of running Azure DevOps agents for pipeline execution.

The thesis is organized into three main parts. The first part sets the context by presenting the concepts and definitions of CI/CD pipelines, as well as the different approaches to application deployment, both in general and in the specific Fujitsu environment.

The second part of the thesis is dedicated to the project itself. It details the project presentation, the target architecture, and the different stages of infrastructure deployment. The emphasis is on writing Infrastructure-as-Code using Terraform, as well as creating CI/CD pipelines to automate the deployment process. The thesis also addresses the construction of Docker images, with a focus on deploying a Self-Hosted-Agent and customizing images to meet the specific project requirements.

The third part of the thesis consists of an assessment. It presents the results achieved through the project, highlighting the success of the test phases and the effectiveness of the deployed agents for pipelines. An analysis is also carried out on the use of Azure DevOps for deployment, identifying the advantages and limitations of this platform. Finally, the thesis highlights my personal contribution to the project and the learning experiences gained. It underscores the significant skill development in areas such as Kubernetes, Helm, Terraform, and cloud infrastructure architecture. The thesis concludes with a deepened understanding of DevOps functionality, acquired by playing the role of intermediary between developers and operations to facilitate and automate processes.

As a whole, the thesis presents a methodical and successful approach to improving the deployment process of the AMCS team at Fujitsu France, demonstrating enhanced efficiency in deployment operations and improved collaboration among teams.

## Introduction

Depuis septembre 2020, maintenant près de deux ans, j'occupe le poste d'apprenti consultant cloud au sein de Fujitsu France. Pour rappel, Fujitsu est une entreprise japonaise spécialisée dans les technologies de l'information et de la communication, comptant environ 140 000 employés répartis dans une centaine de pays à travers le monde. En France, l'entreprise compte plus de 300 employés et se concentre principalement sur la prestation de services informatiques pour d'autres entreprises. Je fais partie de l'équipe AMCS (Applications & Multi-Cloud Services) qui est chargée de tous les projets liés au cloud. Notre objectif est d'aider et d'accompagner les clients de Fujitsu France dans leurs différents projets cloud. En tant qu'ingénieur cloud, mon rôle au sein de cette équipe depuis près de deux ans consiste à travailler sur des sujets Azure et à contribuer à divers projets cloud.

Après une première année d'apprentissage enrichissante, axée principalement sur les bases du Cloud Azure et les architectures applicatives, cette deuxième année s'est révélée encore plus intéressante. Je me sens désormais pleinement intégré à l'équipe et je participe à des projets et des tâches cruciales pour nos clients. Cette année a été fortement axée sur l'apprentissage et le perfectionnement des connaissances et des techniques liées au cloud. J'ai eu l'opportunité de travailler sur des sujets passionnants tels que des déploiements de landing zones entièrement automatisés sur Azure et des plans de reprise d'activité de machines virtuelles dans le cloud. J'ai donc pu renforcer mes compétences sur différentes technologies, ce qui a conduit à me confier un projet individuel visant à améliorer les performances de l'équipe.

Dans ce mémoire, je vais expliquer, schématiser et détailler le projet le plus important sur lequel j'ai travaillé jusqu'à présent en entreprise, en donnant également mon avis. Le projet se concentre principalement sur les technologies et les pratiques DevOps. Il s'agit de déployer un cluster Kubernetes complet sur le cloud pour exécuter des pipelines CI/CD. Dans un premier temps, je vais expliquer ces sujets de manière générale dans le domaine de l'informatique, puis je les relierai au contexte de l'équipe AMCS. Ensuite, j'aborderai tous les aspects techniques de mon projet et j'approfondirai les détails, car c'est un sujet passionnant. Enfin, je présenterai les résultats de ce projet et je donnerai mon avis dans un bilan complet du projet.

## I. Contexte

### 1.1) Pipelines CI/CD : Concepts et définition

Dans le monde du développement logiciel moderne, le concept de DevOps a émergé comme une approche clé pour surmonter les défis liés au déploiement d'applications dans le cloud. Le DevOps vise à combler le fossé entre les équipes de développement et les équipes opérationnelles, en favorisant la collaboration, l'automatisation et l'intégration continue des processus.

Dans ce contexte, les pipelines CI/CD jouent un rôle central. Les pipelines CI/CD sont des ensembles d'étapes automatisées qui permettent de gérer le cycle de vie complet d'une application, depuis le développement initial jusqu'au déploiement en production. Ils permettent aux équipes de développement de garantir une intégration continue et une livraison continue des fonctionnalités, tout en assurant un haut niveau de qualité et de fiabilité.

Les pipelines CI/CD reposent sur deux concepts fondamentaux. Tout d'abord, l'intégration continue (CI) qui consiste à fusionner régulièrement les modifications du code source effectuées par les développeurs dans un référentiel centralisé. Cette pratique permet de détecter rapidement les problèmes d'intégration et de maintenir un code sain et stable. Ensuite, la livraison continue (CD) qui se concentre sur l'automatisation du processus de déploiement, en s'assurant que chaque modification apportée au code puisse être déployée de manière fiable et sans friction. Pour mettre en œuvre les pipelines CI/CD, de nombreux outils populaires sont disponibles. GitLab, GitHub et Azure DevOps sont quelques exemples notables qui offrent des fonctionnalités puissantes pour la gestion des pipelines, la gestion des versions, la gestion des tests et le déploiement continu.

En utilisant ces outils et en adoptant une approche DevOps, les équipes de développement peuvent optimiser leurs cycles de développement, accélérer le déploiement des applications et améliorer la collaboration entre les développeurs et les équipes opérationnelles. Les pipelines CI/CD offrent ainsi une méthode efficace pour automatiser les processus de développement et de déploiement, permettant aux organisations d'innover plus rapidement tout en garantissant une excellente qualité logicielle.

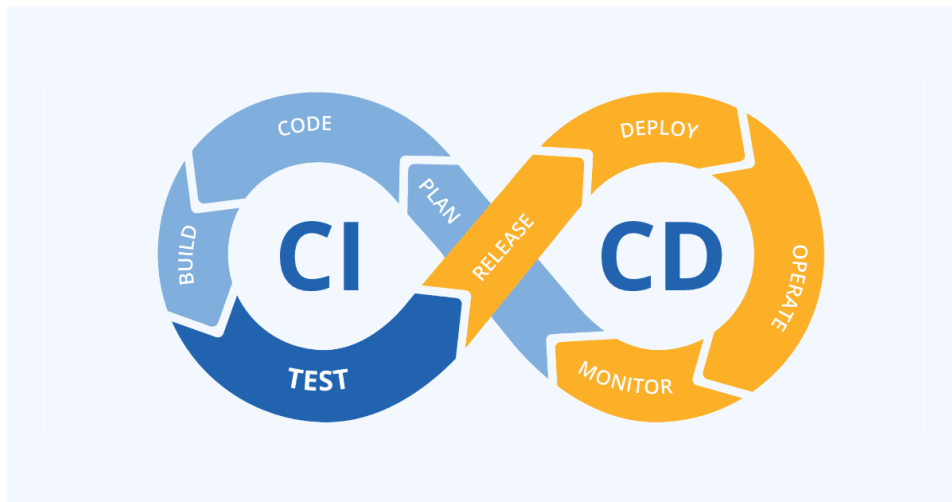


Figure 1 : Fonctionnement de la CI/CD

## 1.2) Le déploiement applicatif

Lorsqu'il s'agit de déployer des applications dans le cloud, les équipes de développement sont confrontées à divers défis qui nécessitent une attention particulière. Au-delà du simple développement de code, la mise en production d'une application nécessite une gestion soignée de l'infrastructure et une prise en compte des différents aspects du déploiement. Je vais donc présenter les défis courants auxquels sont confrontées les équipes de développement lorsqu'elles déploient des applications dans le cloud, avec un accent particulier sur la gestion de l'infrastructure, puis expliquer comment le déploiement applicatif est géré chez AMCS.

### 1.2.1) En général

L'un des principaux défis du déploiement applicatif dans le cloud est la gestion de l'infrastructure. Contrairement aux environnements sur site, où les ressources matérielles sont généralement bien définies et statiques, le cloud offre une flexibilité et une évolutivité pratiquement illimitées. Cependant, cette évolutivité accrue apporte son lot de complexité. Les équipes de développement doivent être en mesure de gérer efficacement les ressources cloud, d'orchestrer les machines virtuelles, de configurer les réseaux, de provisionner les services et de gérer les autorisations.

Un autre défi majeur est la gestion des versions. Avec des cycles de développement rapides et des équipes travaillant simultanément sur différentes fonctionnalités, il est essentiel de mettre en place des mécanismes pour gérer les différentes versions de l'application. Cela inclut la gestion des branches de développement, la gestion des conflits de fusion, le suivi des modifications apportées au code source et la documentation des différentes versions déployées.

La gestion des dépendances constitue également un défi important. Les applications modernes reposent souvent sur une multitude de bibliothèques et de services tiers, ce qui peut compliquer le processus de déploiement. Les équipes de développement doivent être attentives à la gestion des dépendances et s'assurer de leur compatibilité avec les différentes versions de l'application. Des outils tels que les gestionnaires de paquets et les registres de conteneurs peuvent faciliter cette tâche, en assurant la cohérence et la reproductibilité des environnements de déploiement.

La gestion des configurations et la sécurisation des applications déployées sont donc des défis cruciaux. Les équipes doivent mettre en place des mécanismes robustes pour gérer les configurations spécifiques à chaque environnement (développement, test, production) et s'assurer que les secrets, les clés d'API et les informations sensibles sont correctement protégés. La mise en œuvre de bonnes pratiques en matière de sécurité des applications et des infrastructures est essentielle pour prévenir les vulnérabilités et les failles de sécurité.

En conclusion, le déploiement applicatif dans le cloud présente des défis spécifiques liés à la gestion de l'infrastructure, à la gestion des versions, à la gestion des dépendances, à la configuration et à la sécurité. Les équipes de développement doivent être conscientes de ces défis et mettre en place des processus, des outils et des bonnes pratiques adaptés pour garantir un déploiement efficace et sécurisé des applications.

### 1.2.2) Chez Fujitsu

Chez Fujitsu, plus précisément au sein de l'équipe AMCS, le déploiement d'applications dans le cloud revêt une grande importance et constitue un enjeu majeur de nos projets. Nous avons été confrontés à divers déploiements d'applications open-source tels que "Redmine" ou "Owncloud", ainsi qu'à des applications développées par nos clients. Les techniques de déploiement et d'automatisation ont varié en fonction des environnements (conteneurs, machines virtuelles, etc.), des fournisseurs de cloud (Azure, AWS) et surtout des préférences des clients. Certains clients nous laissent le choix de la technique de déploiement, comme ça a été le cas avec Github, tandis que d'autres imposent leurs propres normes de déploiement automatisé, nous obligeant à nous adapter, par exemple en utilisant Gitlab au lieu de Github. Cette diversité de techniques de déploiement rend difficile la création d'une politique de déploiement standardisée et réutilisable pour chaque client.

Cependant, il existe un autre type de projet de déploiement d'application, où les applications sont développées et maintenues par les développeurs de Fujitsu. Au cours des dernières années, AMCS a travaillé en étroite collaboration avec Fujitsu Espagne pour développer des applications à héberger sur Azure, avec Fujitsu Espagne en charge du développement et AMCS des aspects architecturaux et de l'ingénierie cloud. C'est ainsi que des applications importantes et personnalisées ont été créées pour les clients de Fujitsu France.

Cependant, la communication continue avec les développeurs de l'équipe de Fujitsu Espagne pose problème en raison de la distance géographique, de la barrière de la langue et de leurs disponibilités. Cela a conduit à une réorganisation d'AMCS, intégrant dans l'équipe « Cloud & Data Platform » composée d'ingénieurs et d'architectes spécialisés dans AWS et Azure, des développeurs placés sous la responsabilité d'un architecte applicatif. L'objectif de cette nouvelle équipe est de centraliser tous les membres au sein d'une même unité afin de faciliter la collaboration entre l'équipe infrastructure et les développeurs.

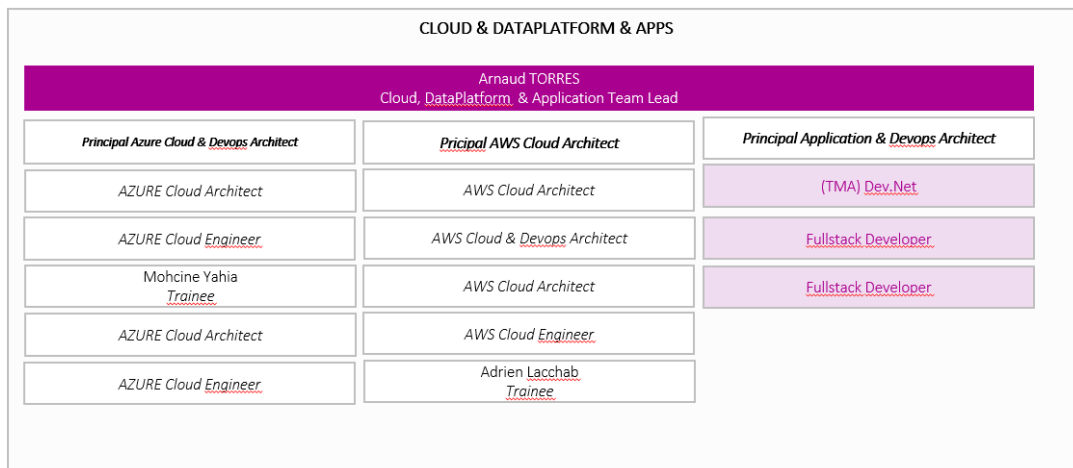


Figure 2 : Organigramme de l'unité Cloud, Dataplatform & Apps de l'équipe AMCS

Cette nouvelle structure nous permet de relever un défi majeur auquel fait face AMCS. En effet, les applications développées et maintenues par Fujitsu sont principalement destinées à une grande entreprise du CAC40. Il est donc essentiel de formaliser une approche basée sur les "squads", qui rassemblent développeurs et architectes travaillant ensemble pour un client donné. Cette approche favorise l'agilité et revêt une importance particulière, car elle permet d'établir un lien fort entre les développeurs et l'infrastructure, favorisant ainsi une approche DevOps.

L'objectif principal de cette nouvelle squad est de développer et de maintenir des applications sur Azure. Il devient donc possible de créer une politique de déploiement standardisée et réutilisable. C'est pourquoi nous avons décidé de nous appuyer sur les technologies Microsoft, telles qu'Azure DevOps, pour mener à bien cette initiative. Je reviendrai plus en détail sur ce point dans la suite de ce mémoire. Dans tous les cas, c'est dans ce contexte DevOps, axé sur l'automatisation, l'optimisation et la collaboration, que j'ai travaillé sur la réalisation de mon projet.



## II. Projet : Déploiement d'un cluster Kubernetes pour la CI/CD

### 2.1) Présentation du Projet

AMCS développe et maintient depuis plusieurs années une application stratégique pour l'un des principaux clients de Fujitsu France. Cette application compte environ 250 utilisateurs par jours et est entièrement déployée sur Azure. Au fil du temps, le processus de déploiement de cette application a évolué, passant d'un déploiement manuel de l'infrastructure Azure à une approche basée sur GitLab et GitHub, sans centralisation du code source. De plus, il n'y avait aucun lien entre le déploiement de l'infrastructure et celui de l'application. Cela a conduit à une refonte complète de l'intégration de l'application dans le cloud, réalisée par l'équipe de développement et l'équipe Azure.

La première étape de cette refonte a consisté à "terraformiser" l'infrastructure existante, c'est-à-dire à déployer l'infrastructure via l'Infrastructure-as-Code (IaC) à l'aide de Terraform, tout en l'améliorant. L'objectif était d'avoir une infrastructure hautement disponible et un plan de reprise d'activité efficace pour l'application. Cette refonte permet également de centraliser le déploiement de l'infrastructure et de l'application sur Azure en utilisant la solution native de Microsoft, Azure DevOps. Azure DevOps est une plateforme de développement logiciel complète qui offre des fonctionnalités pour la gestion du code source, le suivi des problèmes, la planification des projets et l'intégration continue. Elle permet aux équipes de développement de collaborer de manière transparente et de mettre en place des pipelines CI/CD pour le déploiement automatisé des applications.

Actuellement, l'application et son infrastructure sont donc déployées via des pipelines Azure DevOps en utilisant l'agent par défaut de Microsoft. L'agent par défaut dans Azure DevOps est l'agent préconfiguré fourni par Microsoft. Les pipelines sont limités à l'utilisation d'un seul agent, ce qui ne permet pas d'exécuter des tâches en parallèle. De plus, l'agent par défaut de Microsoft contient de nombreux packages inutiles pour l'application, ce qui nécessite l'ajout de tâches supplémentaires d'installation des packages nécessaires, ce qui rallonge considérablement la durée d'exécution des pipelines.

L'objectif de mon projet est d'améliorer ce processus en déployant des agents personnalisés pour cette application et en les mettant à l'échelle de manière automatisée en fonction de la taille des pipelines. Le projet se divise en deux parties : une première axée sur la documentation et les recherches, et une seconde partie purement technique visant à déployer la solution cible.

Avant de se lancer dans les aspects techniques, il est essentiel pour un ingénieur de réfléchir à la solution la plus optimisée possible. Dans ce cadre, des discussions avec le développeur de l'application et l'ingénieur Azure permettront de déterminer les packages nécessaires pour les agents personnalisés que je vais fournir. Sur le plan technique, il sera nécessaire de déployer un cluster Kubernetes sur Azure et des images Docker personnalisées pour permettre aux agents de s'exécuter dans un environnement sécurisé et hautement disponible. Enfin, le dernier défi consistera à anticiper les projets futurs afin que toutes les applications natives d'Azure puissent être déployées de la même manière, en utilisant des agents Azure DevOps déployés sur le même cluster Kubernetes.

## 2.2) Architecture cible

Le projet est d'une grande complexité, il est donc essentiel de le représenter visuellement à l'aide d'un schéma décrivant son architecture et les dépendances entre les différentes ressources. Avant d'aborder les aspects techniques, il est important de présenter en détail l'architecture globale du projet. L'architecture repose sur un cluster Azure Kubernetes privé qui sera interconnecté à l'organisation Azure DevOps, où les pipelines sont définis. De plus, il est nécessaire de connecter ce cluster aux réseaux des projets associés à ces pipelines afin de pouvoir les déployer.

Dans cette section, je vais détailler chaque composant de l'architecture du projet. Nous commencerons par explorer la partie Azure DevOps, en exposant les fonctionnalités clés et leur relation avec les pipelines. Ensuite, nous nous pencherons sur le cœur de l'infrastructure Azure que je vais provisionner, en décrivant les principaux services et ressources utilisés. Enfin, nous aborderons la question du peering avec les autres projets, soulignant l'importance de l'interconnectivité entre les différents environnements.

Il est important de noter que le projet est en cours de réalisation, et certaines des configurations et fonctionnalités mentionnées dans cette architecture peuvent ne pas encore être mises en place.

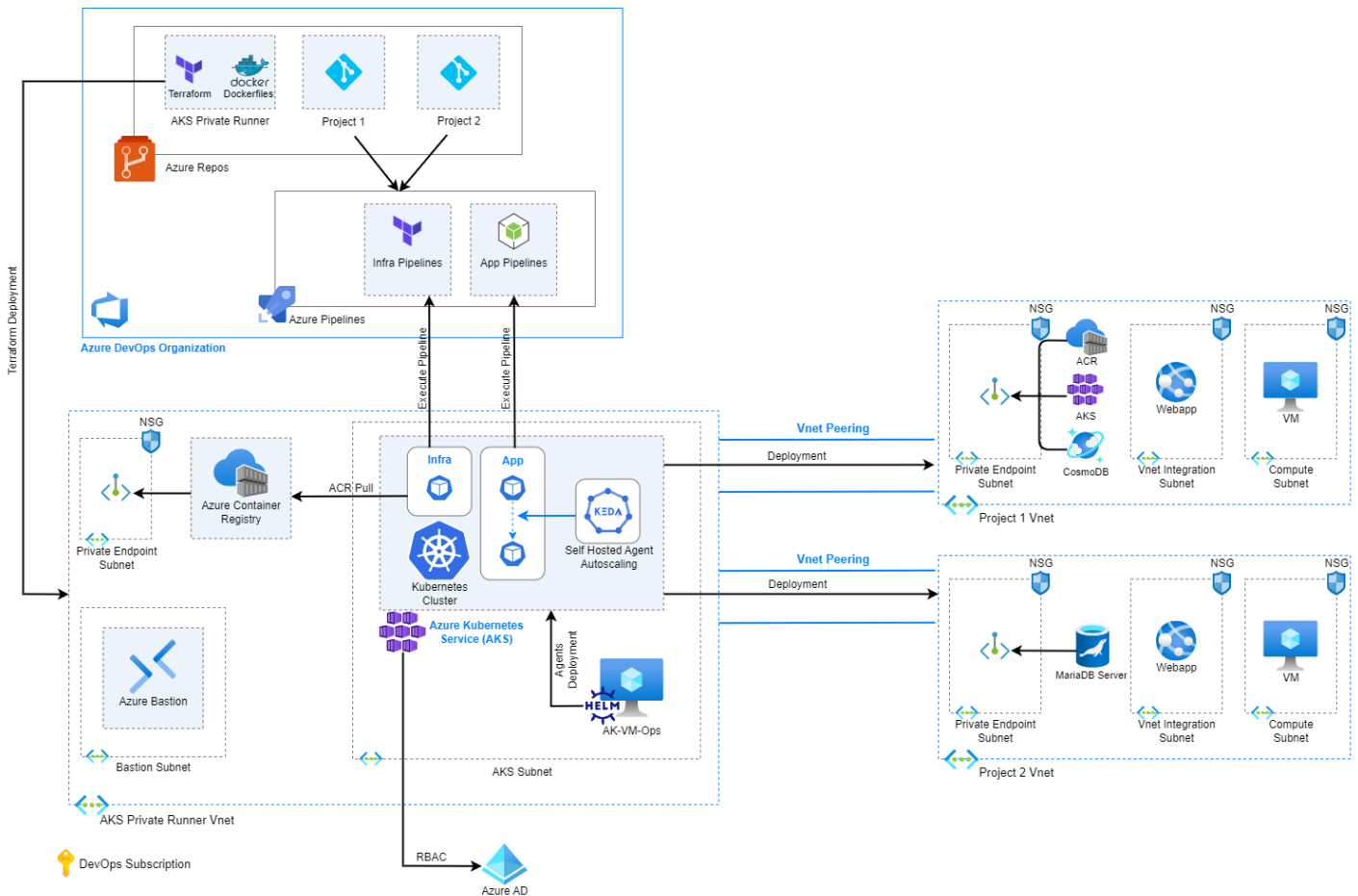


Figure 3 : Schéma d'architecture du projet

### 2.2.1) Azure DevOps : Vue d'ensemble



#### ➤ Organisation Azure DevOps

Notre premier focus se fait sur l'organisation Azure DevOps. Une organisation Azure DevOps est une plateforme collaborative qui facilite la gestion du cycle de vie du développement logiciel. Elle regroupe tous les projets de déploiement applicatif sur Azure pour les clients de Fujitsu France. Chaque projet sur Azure DevOps est une entité distincte qui regroupe les ressources nécessaires pour le déploiement d'une application spécifique. Cependant, il convient de noter qu'un projet de déploiement peut comporter plusieurs projets Azure DevOps associés.

Dans notre cas, nous avons plusieurs projets liés à chaque application déployée, répartis généralement en deux catégories. Tout d'abord, il y a des projets dédiés au développement, utilisés exclusivement par les développeurs. Ces projets centralisent le code source et les processus de déploiement de l'application. Ensuite, il y a des projets infrastructure, utilisés par les ingénieurs cloud, y compris moi-même. Ces projets sont destinés au stockage et au déploiement des scripts d'Infra-As-Code, principalement basés sur Terraform.



### ➤ Azure Repos

Dans ces projets Azure DevOps, nous utilisons des dépôts (Azure Repos). Les dépôts sont des espaces où le code source est stocké et géré. Pour la partie application, les dépôts contiennent le code source du backend et du frontend. Pour les projets d'infrastructure, les dépôts contiennent le code Terraform permettant de déployer les infrastructures des applications. Dans notre cas, nous avons un dépôt spécifique appelé "AKS-Private-Runner" qui a été le principal focus tout au long du projet. Ce dépôt comprend le code Terraform de l'infrastructure dont je parlerai ultérieurement, les configurations Docker des agents Azure DevOps, ainsi que les "charts" Kubernetes, un point qui sera abordé plus tard dans ce mémoire. Ces dépôts exécutent des pipelines Azure, qui jouent un rôle clé dans le processus de déploiement.



### ➤ Azure Pipelines

Le dernier aspect de l'organisation Azure DevOps concerne donc les pipelines. Les pipelines Azure DevOps sont des flux de travail automatisés qui permettent de gérer et de contrôler le déploiement d'une application. Ils sont constitués de différentes étapes (stages) et tâches (jobs) qui permettent d'effectuer des actions spécifiques, telles que la compilation, les tests et le déploiement. Dans les projets de l'organisation AMCS, chaque dépôt possède un pipeline qui permet le déploiement du code. Les pipelines côté applicatif sont utilisés pour initialiser les environnements de développement et pour exécuter le code. Ils nécessitent donc des environnements comprenant des langages spécifiques tels que Python, Node.js, etc. Les pipelines côté infrastructure sont composés d'instructions permettant d'initialiser et de déployer l'infrastructure en tant que code (Infra-As-Code). Ils s'exécutent donc sur des environnements comprenant des outils tels que Terraform, Azure CLI, Docker, etc.

Ce sont les agents Azure DevOps qui exécutent ces pipelines, et ils sont regroupés dans des "agent pools". Un agent Azure DevOps est un composant logiciel qui permet d'exécuter les tâches définies dans les pipelines. Les agents sont installés sur des machines virtuelles ou physiques et sont configurés pour se connecter à l'organisation Azure DevOps.

Les "agent pools" sont des groupes de ressources dans Azure DevOps qui regroupent plusieurs agents. Ils permettent de gérer et d'organiser les agents en fonction de critères spécifiques, tels que leur capacité de traitement, leur emplacement géographique ou leur type d'environnement. Les pipelines seront exécutés sur les agents personnalisés dont je parlerai

par la suite. Cependant, le dépôt "AKS-Private-Runner" est responsable de la création de l'infrastructure de ces agents, il doit donc s'exécuter sur l'agent par défaut fourni par Microsoft.

### 2.2.2) L'infrastructure requise



#### ➤ **Souscriptions Azure**

Lors de la phase de conception de cette infrastructure, dites phase de « design », l'objectif principal qui en est ressorti est que le cluster Kubernetes doit centraliser tous les agents pour tous les projets applicatifs, et donc qu'il doit être au centre de toutes les autres infrastructures. Il est essentiel de concevoir l'infrastructure de manière intelligente dès les couches supérieures d'Azure, en commençant par la souscription Azure. Une souscription Azure est une unité d'organisation qui regroupe des ressources et des services Azure et permet de gérer les coûts et les accès. Dans notre cas, nous avons créé une souscription dédiée à l'infrastructure des agents Azure DevOps, appelée "DevOps". L'idée est que tous les autres projets, situés dans d'autres souscriptions, puissent se connecter à cette nouvelle souscription DevOps pour obtenir un agent de déploiement.



#### ➤ **Azure Virtual Network**

Le premier aspect à prendre en compte dans une infrastructure cloud est bien sûr le réseau. Dans notre infrastructure, nous utilisons un seul réseau virtuel Azure (VNet). Un VNet est un réseau isolé et configurable qui permet aux ressources Azure de communiquer entre elles. Ce VNet servira de réseau principal pour tous les composants de l'infrastructure, et il sera également le lien permettant aux autres réseaux de s'apparier avec notre cluster Kubernetes. Le VNet est composé de trois sous-réseaux : un pour le cluster Kubernetes, un pour le registre de conteneurs et un pour le serveur Bastion. Chaque sous-réseau joue un rôle spécifique que nous détaillerons lors de l'explication de chaque composant.



#### ➤ **Azure Container Registry**

Faisons maintenant un focus sur chaque élément de cette infrastructure. Commençons par « l'Azure Container Registry » (ACR) L'ACR est un service Azure qui permet de stocker des images Docker. Cette ACR va donc permettre de stocker toutes les images Docker construites pour chaque projet. C'est un élément central, car la première étape pour chaque nouveau projet sera de déployer un agent Azure DevOps personnalisé, et Kubernetes a besoin d'un endroit où les images Docker sont stockées pour pouvoir déployer l'agent. Par défaut sur Azure, un ACR est public, ce qui signifie qu'il est accessible depuis n'importe quel réseau tant que l'accès est autorisé. Afin de sécuriser l'accès à l'ACR, nous le configurons pour qu'il soit lié à un "private endpoint", qui est situé dans un sous-réseau distinct. Un private

endpoint permet d'accéder à un service Azure via une connexion privée, sans exposer le service sur Internet, ce qui renforce la sécurité de l'ACR et restreint l'accès à celui-ci.



## ➤ Azure Kubernetes Services

Au cœur de notre projet, on retrouve « Azure Kubernetes Services » (AKS). AKS est un service Azure qui simplifie le déploiement, la gestion et l'évolutivité des clusters Kubernetes. Kubernetes est un système open-source de gestion et d'orchestration de conteneurs qui permet de déployer, mettre à l'échelle et gérer des applications conteneurisées de manière efficace et fiable. Dans notre cas, l'objectif final est de déployer un cluster AKS privé dans son sous-réseau dédié. Ce cluster sera responsable de l'exécution des agents Azure DevOps pour tous les projets applicatifs.

Étant donné que les agents Azure DevOps sont des charges de travail légères, un seul pool de nœuds (node pool) est suffisant. Un node pool est un groupe de machines virtuelles identiques qui exécutent des tâches spécifiques dans un cluster Kubernetes. Dans notre cas, le node pool sera composé d'un ensemble de machines virtuelles à mise à l'échelle automatique (VMSS - Virtual Machine Scale Set). Un VMSS est une fonctionnalité d'Azure qui permet de créer et de gérer automatiquement un groupe de machines virtuelles identiques, ajustées en fonction de la demande de charge de travail. Pour garantir la haute disponibilité (HA) du cluster AKS, nous configurons le VMSS pour qu'il répartisse les machines virtuelles sur trois zones de disponibilité différentes. Les zones de disponibilité sont des emplacements physiques distincts au sein d'une région Azure, offrant une redondance et une résilience accrues en cas de panne d'une zone spécifique. En répartissant les machines virtuelles sur plusieurs zones de disponibilité, nous assurons la continuité des opérations en cas de défaillance d'une zone.

Dans un cluster Kubernetes, plusieurs composants interagissent pour permettre le déploiement et la gestion des applications. Parmi les principaux composants, nous retrouvons les namespaces, les pods, les deployments et les secrets, qui sont étroitement liés à ce que nous avons déployé. Un namespace est une façon de regrouper et d'isoler logiquement les ressources au sein d'un cluster Kubernetes. Dans notre cas, nous avons choisi de regrouper tous les agents dans un même namespace nommé "devopsagent". Cette approche permet une meilleure organisation et gestion des ressources liées aux agents Azure DevOps. Nous expliquerons plus en détail comment nous avons séparé les différents agents pour optimiser l'utilisation des ressources. Les agents Azure DevOps sont en réalité des pods au sein du cluster. Un pod est la plus petite unité déployable dans Kubernetes. Chaque pod représente une instance d'un agent Azure DevOps et est configuré pour se connecter aux agent pools d'Azure DevOps. Ce sont sur ces pods que les pipelines seront exécutés, permettant ainsi le déploiement des applications. Nos pipelines d'Infrastructure-as-Code n'exécutent pas de tâches en parallèle, donc un seul pod d'infrastructure par projet est suffisant. Cependant, dans le cas des pipelines applicatifs, la mise à l'échelle devient importante car ces pipelines exécutent des tâches en parallèle. Pour cela, nous utilisons le plugin Keda, que nous détaillerons par la suite. Grâce à Keda, les pods applicatifs peuvent être mis à l'échelle en

fonction du nombre de tâches en attente dans la file d'attente Azure Pipeline. Cette fonctionnalité permet d'optimiser l'utilisation des ressources en adaptant la capacité de traitement en fonction de la charge de travail. Ainsi, c'est dans cet environnement, au sein du cluster Kubernetes, que j'ai réalisé la majeure partie de mon travail.



### ➤ **Azure Virtual Machine**

Pour accéder et configurer notre cluster AKS dans un réseau privé, il est nécessaire de provisionner une machine virtuelle dans le même sous-réseau. Cette machine jouera un rôle essentiel dans la gestion de l'infrastructure Kubernetes et des images Docker. Dans notre cas, j'ai opté pour une machine virtuelle Linux. Le choix d'une machine virtuelle Linux repose sur plusieurs raisons. Tout d'abord, les environnements Kubernetes et Docker sont plus adaptés aux systèmes Unix, offrant une meilleure expérience d'utilisation et une plus grande légèreté. De plus, les outils en ligne de commande tels que Docker et kubectl sont plus couramment utilisés dans les environnements Unix. En utilisant une machine virtuelle Linux, nous pouvons tirer pleinement parti de ces outils pour gérer efficacement notre infrastructure. J'ai opté pour une machine virtuelle Ubuntu, car c'est la distribution Linux que je maîtrise le mieux.

La machine virtuelle aura pour rôle de réaliser l'Infrastructure-as-Code pour Kubernetes en déployant les agents Azure DevOps. Pour faciliter ce processus, nous utiliserons Helm, un gestionnaire de packages pour Kubernetes. Helm simplifie le déploiement, la mise à jour et la gestion des applications sur Kubernetes en utilisant des "charts", qui sont des packages préconfigurés contenant les ressources nécessaires. Nous détaillerons plus en profondeur Helm dans la suite de ce mémoire. Ainsi, grâce à cette VM Linux, nous pourrons effectuer l'IaC pour Kubernetes, déployer les agents Azure DevOps et utiliser Helm pour faciliter la gestion de notre cluster AKS.



### ➤ **Azure Bastion**

Pour accéder à la machine virtuelle de configuration d'AKS, qui est, comme expliqué précédemment, privée, nous utilisons un serveur bastion. Un serveur bastion est une solution de saut d'accès sécurisée qui permet d'accéder à des machines virtuelles privées via une interface Web. Dans Azure, on retrouve Azure Bastion qui est le service de passerelle d'accès sécurisé offrant une connectivité RDP (Remote Desktop Protocol) et SSH (Secure Shell) basée sur le navigateur directement depuis le portail Azure. Il permet d'accéder en toute sécurité aux machines virtuelles privées sans avoir à exposer les ports RDP ou SSH sur Internet public. Dans notre cas, nous allons utiliser Azure Bastion pour accéder à la machine virtuelle privée en SSH. Il est important de noter qu'Azure Bastion est présent dans son propre sous-réseau dédié, conformément à la norme d'Azure qui prévoit que le bastion soit la seule ressource de son sous-réseau et que celui-ci porte toujours le nom "AzureBastionSubnet". Cela garantit une isolation appropriée et renforce la sécurité de l'accès à la machine virtuelle de configuration d'AKS.



### Azure Active Directory (renommé récemment en Entra ID)

Le dernier composant de l'infrastructure dont nous n'avons pas encore abordé est Azure Active Directory (Azure AD). Azure AD est un service d'annuaire basé sur le cloud fourni par Microsoft. Il s'agit d'une solution de gestion des identités et des accès qui permet de sécuriser et de gérer l'accès aux ressources cloud et aux applications. Dans notre infrastructure, Azure AD joue un rôle crucial car il gère les droits d'accès et les autorisations des différentes ressources et des comptes de service Azure. Pour pouvoir déployer une image sur l'Azure Container Registry (ACR), le compte de service utilisé par les scripts de l'ingénieur cloud (moi-même) doit avoir le rôle lui permettant de pousser une image dans le registre. D'autre part, Azure Kubernetes Service (AKS) récupère les images Docker à partir de l'ACR et doit donc disposer du rôle lui permettant de "pull" les images Docker depuis l'ACR. Azure AD permet de gérer ces autorisations et de garantir que seules les entités appropriées ont les droits nécessaires pour accéder aux ressources.

Dans un contexte sécurisé tel que le nôtre, Azure AD est essentiel pour assurer une gestion centralisée des identités, des rôles et des autorisations, garantissant ainsi la sécurité et la conformité de l'infrastructure.

#### 2.2.3) Interconnexion avec les projets applicatif

Lors de la conception de l'architecture présentée, il est important de noter l'inclusion de deux projets d'application. Cela implique que pour chaque projet, nous avons deux projets Azure DevOps (un pour l'application et un pour l'infrastructure en tant que code), deux types d'agents personnalisés dans le cluster Kubernetes, ainsi qu'une mise en correspondance des réseaux virtuels entre le Vnet de l'AKS et le Vnet de l'infrastructure de l'application. L'appairage de Vnet, également appelé peering, est utilisé pour permettre la communication entre ces différents environnements virtuels. Il est important de noter que le peering ne doit pas être transitif, ce qui signifie que les projets 1 et 2 sont appairés avec l'AKS, mais les projets 1 et 2 ne communiquent en aucun cas entre eux, car ce sont deux applications distinctes, voire même pour des clients différents.

L'intérêt du peering réside dans la capacité à déployer depuis un environnement privé, en l'occurrence l'AKS, vers une infrastructure également privée, comme le montre le schéma avec les trois types de sous-réseau représentés : le sous-réseau du private endpoint, celui de l'intégration Vnet, et enfin celui du compute. Ces différents types de sous-réseau sont utilisés pour des besoins spécifiques, tels que le subnet private endpoint pour le registre de conteneurs ou les bases de données, le subnet Vnet integration pour l'application Web, et le subnet compute pour les machines virtuelles.



Dans ce mémoire, je vais me concentrer sur l'intégration du cluster AKS pour le projet 1 uniquement. Cependant, il est important de souligner que l'objectif de cette architecture est de pouvoir intégrer ces méthodes de déploiement pour tous nos projets futurs. Ainsi, nous pourrions imaginer que les projets 3, 4, 5, etc., viennent se connecter à ce même AKS avec leurs propres agents Azure DevOps. Cette architecture met donc l'accent sur la facilité d'intégration de nos futurs projets applicatifs à notre cluster privé.

La réalisation du schéma a été une étape très intéressante, car il s'agit de la première étape d'un projet complexe. La visualisation graphique de l'architecture permet de mieux appréhender les différentes composantes et leurs interactions. De plus, cela facilite la communication avec les parties prenantes du projet en fournissant une représentation visuelle claire. Pour finir, je souhaite donner un conseil aux personnes habituées à créer des schémas. L'extension Draw.io pour Visual Studio Code est très pratique, car elle permet de créer des schémas à l'aide du célèbre logiciel open source drawio directement dans un onglet de VS Code. Cela permet de modifier et de mettre à jour facilement les schémas tout en travaillant sur la partie technique, comme dans mon cas avec Terraform, Docker ou Kubernetes.

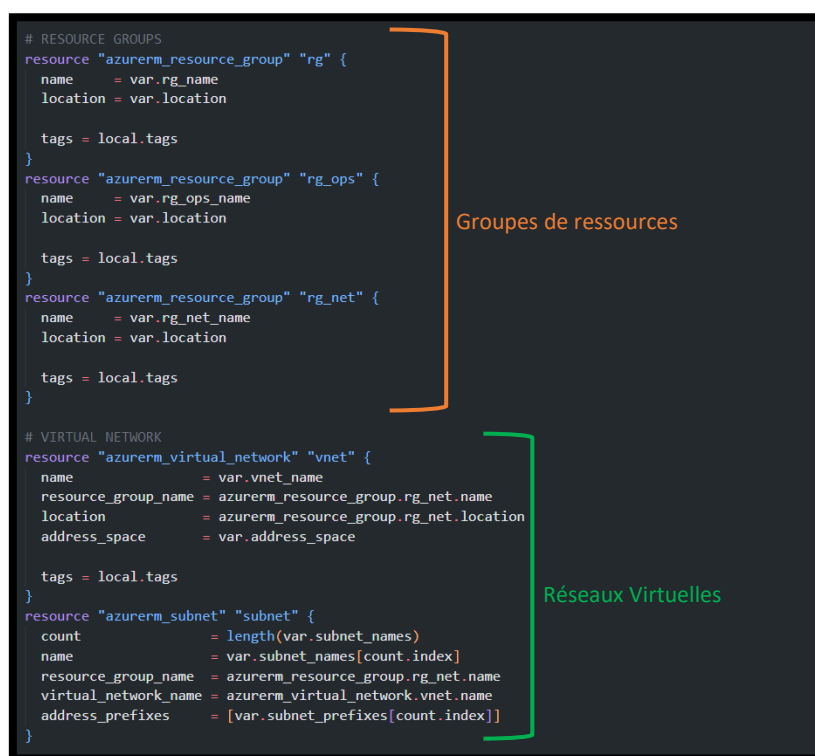
### 2.3) Déploiement de l'infrastructure

La première étape technique essentielle de notre projet consiste à déployer l'infrastructure cible. Dans le contexte du cloud, il est devenu la norme de créer une infrastructure immuable en utilisant l'approche de l'Infrastructure-as-Code (IaC). Bien que l'infrastructure cible puisse rarement changer, il est crucial de l'aborder avec les meilleures pratiques du déploiement cloud, du codage Terraform à l'automatisation du déploiement via les pipelines CI/CD.

Dans cette section, nous allons explorer en détail le processus que j'ai suivi, de la rédaction de l'Infrastructure-as-Code à la mise en œuvre de déploiements automatisés via les pipelines Azure DevOps. Il est important de noter que la configuration Terraform que nous présentons ici est une première version et que des améliorations sont en cours, notamment en ce qui concerne l'isolation de l'Azure Container Registry via un private endpoint. Ces configurations sont aujourd'hui toujours en cours d'amélioration pour garantir une infrastructure fiable et sécurisée.

### 2.3.1) Rédaction de l'Infrastructure-as-Code

La rédaction d'un code Terraform peut parfois être complexe, nécessitant des compétences de développement avancées pour utiliser des fonctions telles que des boucles et des mappings, surtout lorsqu'il implique la création de plusieurs ressources liées. Cependant, dans notre cas, l'infrastructure à déployer est relativement "simple". Elle se compose de quelques groupes de ressources, un réseau virtuel découpé en plusieurs sous-réseaux, un Azure Container Registry (ACR) et un Azure Kubernetes Service (AKS) ayant des droits sur le registre de conteneur, une machine virtuelle de gestion et un Azure Key Vault permettant de stocker les secrets de la VM et du cluster Kubernetes. Étant donné le nombre limité de ressources, il n'a pas été nécessaire de découper le Terraform en plusieurs modules, mais cela peut être judicieux pour les projets plus complexes. Un module Terraform est une abstraction réutilisable d'une partie de votre infrastructure. Il permet de séparer le code en blocs indépendants et de faciliter la gestion et la réutilisation du code. Dans notre cas j'ai tout développé dans un même fichier Terraform regroupant toutes les ressources.



```
# RESOURCE GROUPS
resource "azurerm_resource_group" "rg" {
  name     = var.rg_name
  location = var.location

  tags = local.tags
}
resource "azurerm_resource_group" "rg_ops" {
  name     = var.rg_ops_name
  location = var.location

  tags = local.tags
}
resource "azurerm_resource_group" "rg_net" {
  name     = var.rg_net_name
  location = var.location

  tags = local.tags
}

# VIRTUAL NETWORK
resource "azurerm_virtual_network" "vnet" {
  name                = var.vnet_name
  resource_group_name = azurerm_resource_group.rg_net.name
  location             = azurerm_resource_group.rg_net.location
  address_space       = var.address_space

  tags = local.tags
}
resource "azurerm_subnet" "subnet" {
  count                = length(var.subnet_names)
  name                 = var.subnet_names[count.index]
  resource_group_name = azurerm_resource_group.rg_net.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = [var.subnet_prefixes[count.index]]
}
```

Figure 4 : Extrait du Terraform : Ressource Groupes et Réseaux Virtuel

Commençons par expliquer les aspects simples mais essentiels de l'infrastructure, à savoir les groupes de ressources et le réseau virtuel. Dans notre code Terraform, nous avons créé trois groupes de ressources distincts : un pour les ressources principales, un pour les opérations (contenant la machine de gestion), et un pour le réseau (regroupant le réseau virtuel et le bastion). Le réseau virtuel est défini à l'aide de la ressource "azurerm\_virtual\_network", où nous utilisons la fonction count pour créer autant de sous-réseaux que spécifiés dans la variable de sous-réseaux. Cela permet de découper le réseau virtuel en différentes zones pour une meilleure organisation et sécurité.

Passons maintenant à l'aspect central de notre projet, à savoir l'Azure Container Registry (ACR) et l'Azure Kubernetes Service (AKS). Pour le moment, nous n'avons pas pris en compte le private endpoint du registre, ce qui rend leur création relativement simple. Pour le cluster AKS, nous devons prendre en compte certains paramètres, comme l'attribution de l'identité "SystemAssigned" pour le cluster afin de pouvoir donner les droits nécessaires. Nous avons également configuré le profil réseau pour utiliser le réseau virtuel créé précédemment. Enfin, nous devons donner le rôle "AcrPull" à l'AKS pour lui permettre de récupérer des images depuis l'ACR.

```
# CONTAINER REGISTRY & KUBERNETES CLUSTER
resource "azurerm_container_registry" "acr" {
  name                = var.acr_name
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  sku                 = "Standard"

  tags = local.tags
}

resource "azurerm_kubernetes_cluster" "aks" {
  name                = var.aks_name
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  private_cluster_enabled = true
  dns_prefix_private_cluster = var.aks_name

  network_profile {
    network_plugin = "azure"
  }

  default_node_pool {
    name                = "agentpool"
    node_count          = 1
    vm_size             = "Standard_DS2_v2"
    type               = "VirtualMachineScaleSets"
    zones              = [1, 2, 3]
    enable_auto_scaling = false
    vnet_subnet_id     = azurerm_subnet.subnet.0.id
  }

  identity {
    type = "SystemAssigned"
  }

  tags = local.tags
}

resource "azurerm_role_assignment" "role_acrpull" {
  scope                = azurerm_container_registry.acr.id
  role_definition_name = "AcrPull"
  principal_id         = azurerm_kubernetes_cluster.aks.identity.0.principal_id
}
```

Figure 5 : Extrait du Terraform : ACR et AKS

Enfin, une autre partie intéressante de notre code Terraform concerne le provisionnement de la machine virtuelle de gestion (VM ops). Pour garantir la sécurité lors du provisionnement de la VM, nous créons d'abord un Azure Key Vault. Nous autorisons ensuite le service principal utilisé pour déployer Terraform à gérer les secrets de ce Key Vault. L'intérêt est de pouvoir stocker directement les informations d'identification de la VM dans le Key Vault, sans que l'ingénieur cloud (moi-même) ne les voie directement ou que ces informations soient vulnérables en étant stockées directement dans les variables du code Terraform. Une fois cela fait, nous déployons une machine virtuelle Ubuntu 18.04 standard avec les informations d'identification provenant du Key Vault. Nous utilisons également une option appelée "custom\_data" pour fournir un script cloud-init qui permet d'installer automatiquement les outils nécessaires, tels que Azure CLI, Docker, kubectl et Helm, sur la VM (voir [annexe](#)/Terraform pour le script du custom data).

```
resource "azurerm_key_vault_secret" "vm_ops_admin_password" {
  name       = "vm-ops-admin-password"
  value      = random_password.vm_ops_admin.result
  key_vault_id = azurerm_key_vault.kv.id
}
resource "azurerm_key_vault_secret" "vm_ops_admin_username" {
  name       = "vm-ops-admin-username"
  value      = var.vm_username
  key_vault_id = azurerm_key_vault.kv.id
}
resource "azurerm_linux_virtual_machine" "vm_ops" {
  name                        = var.vm_name
  resource_group_name       = azurerm_resource_group.rg_ops.name
  location                  = azurerm_resource_group.rg_ops.location
  size                      = var.vm_size
  admin_username            = azurerm_key_vault_secret.vm_ops_admin_username.value
  admin_password            = azurerm_key_vault_secret.vm_ops_admin_password.value
  network_interface_ids    = [azurerm_network_interface.vm_ops_nic.id]
  disable_password_authentication = false

  os_disk {
    name           = "disk-${var.vm_name}"
    caching        = "ReadWrite"
    storage_account_type = "Standard_LRS"
  }

  source_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "18.04-LTS"
    version   = "latest"
  }

  identity {
    type = "SystemAssigned"
  }

  custom_data = filebase64("cloud-init/customdata.sh")

  tags = local.tags
}
```

Génération des credentials de la VM et stockage dans le Key Vault

Figure 6 : Extrait du Terraform : VM d'opérations

La rédaction de l'Infrastructure-as-Code pour mon projet a donc été un processus essentiel pour déployer l'infrastructure cible de manière automatisée, robuste et contrôlée. Je vais maintenant aborder la création des pipelines CI/CD, qui jouent un rôle crucial dans l'exécution et le déploiement automatisé du code Terraform.

### 2.3.2) Création des pipelines CI/CD

Après avoir rédigé le code Terraform, il faut l'exécuter pour déployer l'infrastructure. Bien qu'il soit possible d'exécuter du Terraform depuis son poste de travail, cette pratique n'est pas recommandée dans un environnement professionnel. En effet, l'utilisation des pipelines CI/CD offre de nombreux avantages pour le déploiement de l'infrastructure et garantit une approche plus robuste et reproductible.

L'une des raisons majeures d'utiliser des pipelines CI/CD pour exécuter du Terraform réside dans la gestion de l'état de l'infrastructure. L'état, ou tfstate, est un fichier crucial qui permet de garder une trace de l'état actuel des ressources déployées. Il indique notamment quelles ressources ont été créées, modifiées ou supprimées. Pour garantir la cohérence et la sécurité de l'infrastructure, il est primordial de stocker cet état de manière centralisée. C'est là qu'intervient le backend de stockage, qui offre un emplacement sécurisé pour conserver le tfstate. Dans notre cas, j'ai préalablement provisionné un compte de stockage Azure qui servira à stocker les fichiers tfstate.

En gardant Azure DevOps comme point central de déploiement, nous nous assurons de conserver un environnement intégré pour tous nos projets. J'ai donc rédigé un pipeline Azure DevOps spécifique pour le déploiement du code Terraform responsable de la création du cluster Kubernetes. Pour cette étape, nous utilisons les agents par défaut fournis par Microsoft, car d'autres options ne sont pas envisageables pour ce type de déploiement. Le pipeline Azure DevOps que j'ai mis en place garantit l'exécution automatisée et cohérente du code Terraform. Il assure également la gestion du tfstate, en s'assurant qu'il est stocké de manière sécurisée dans le backend de stockage Azure préalablement provisionner. Grâce à cette approche, on bénéficie d'une traçabilité complète de l'état de l'infrastructure et de la reproductibilité des déploiements à travers des pipelines CI/CD bien structurés.

```
- task: TerraformTaskV4@4
  displayName: Terraform Plan
  inputs:
    provider: "azurerm"
    command: "plan"
    workingDirectory: "$(System.DefaultWorkingDirectory)/Terraform"
    commandOptions: '-out "planfile"'
    environmentServiceNameAzureRM: " "

- task: PublishBuildArtifacts@1
  displayName: Upload planfile
  inputs:
    PathToPublish: "$(System.DefaultWorkingDirectory)/Terraform/"
    ArtifactName: "drop"
    publishLocation: "Container"

- job: manual_intervention
  displayName: Wait for terraform plan validation
  dependsOn: terraform_plan
  pool: server
  timeoutInMinutes: 4320 # job times out in 3 days
  steps:
    - task: ManualValidation@0
      inputs:
        notifyUsers: " "
        instructions: |
          you should validate the Terraform Plan file
          $(Build.BuildId)

- stage: Terraform_CD

  pool:
    vmImage: ubuntu-latest

  jobs:
    - job: terraform_apply

      steps:
        - task: DownloadBuildArtifacts@0
          displayName: Download planfile
          inputs:
            buildType: "current"
            downloadType: "specific"
            itemPattern: "drop/planfile"
            downloadPath: "$(System.ArtifactsDirectory)"

        - task: CopyFiles@2
          displayName: Copy planfile
          inputs:
            SourceFolder: "$(System.ArtifactsDirectory)/drop"
            Contents: "planfile"
            TargetFolder: "$(System.DefaultWorkingDirectory)/Terraform"
```

Figure 7 : Extrait du pipeline Azure DevOps déployant le Terraform

Dans la configuration du pipeline, j'ai fait le choix d'une division en deux stages distincts : « **Terraform\_CI** » et « **Terraform\_CD** ». Cette division correspond à la séparation classique entre l'intégration continue (CI) et le déploiement continu (CD). La première étape, **Terraform\_CI**, est dédiée à la validation du code Terraform et à la génération d'un plan d'exécution. Elle s'assure que le code est correct et qu'il n'y a pas d'erreurs potentielles avant le déploiement. La deuxième étape, **Terraform\_CD**, est responsable de l'application réelle du plan d'exécution généré précédemment, ce qui signifie le déploiement réel des ressources dans l'environnement cible. Cette division en deux étapes distinctes garantit un processus de déploiement robuste et bien contrôlé. Pour assurer un contrôle plus rigoureux du processus de déploiement, une étape supplémentaire de validation manuelle (**manual\_intervention**) est introduite à la fin du **Terraform\_CI**. Cette étape est conçue pour permettre une validation humaine avant que le déploiement réel **Terraform\_CD** ne soit effectué. L'architecte cloud ou toute autre personne responsable peut examiner le plan d'exécution généré et donner son approbation avant de poursuivre le déploiement. Cela garantit une étape de vérification supplémentaire pour s'assurer que le déploiement se déroule sans accroc et minimise les risques liés à des erreurs potentielles.

Dans chaque job du pipeline, on spécifie un agent par défaut Ubuntu (**vmImage: ubuntu-latest**) pour exécuter les différentes tâches. Un agent est une machine virtuelle ou un conteneur qui exécute les étapes du pipeline. L'utilisation d'un agent par défaut Ubuntu signifie que les tâches du pipeline s'exécuteront dans un environnement basé sur Ubuntu. Les tâches Azure DevOps spécifiées dans le pipeline (**TerraformTaskV4**, **DownloadBuildArtifacts**, **CopyFiles**, **ManualValidation**, etc.) sont des étapes automatisées qui exécutent des actions spécifiques. Par exemple, la tâche **TerraformTaskV4** est une tâche personnalisée qui permet d'exécuter des commandes Terraform. Elle est utilisée dans plusieurs étapes pour initialiser Terraform, valider le code, créer un plan d'exécution, et enfin, appliquer ce plan pour déployer l'infrastructure.

Dans la première étape **Terraform\_CI**, on effectue plusieurs tâches, dont la validation du code Terraform à l'aide de la tâche **TerraformTaskV4**. Une fois que le code est validé sans erreur, la tâche **TerraformPlan** est utilisée pour créer un fichier de plan (**planfile**) qui contient les détails du déploiement à effectuer. Ce fichier de plan est généré en utilisant l'option **-out** lors de la commande **Terraform plan**. Le **planfile** est ensuite publié en tant qu'artifact à l'aide de la tâche **PublishBuildArtifacts**. Cela permet de stocker le plan d'exécution généré dans l'artifacts de la build, afin qu'il soit disponible pour les étapes ultérieures du pipeline, notamment lors de l'étape **Terraform\_CD** où le déploiement réel aura lieu. L'étape **Terraform\_CD** télécharge le **planfile** à l'aide de la tâche **DownloadBuildArtifacts**, puis l'applique pour effectuer le déploiement réel avec la tâche **TerraformApply**.

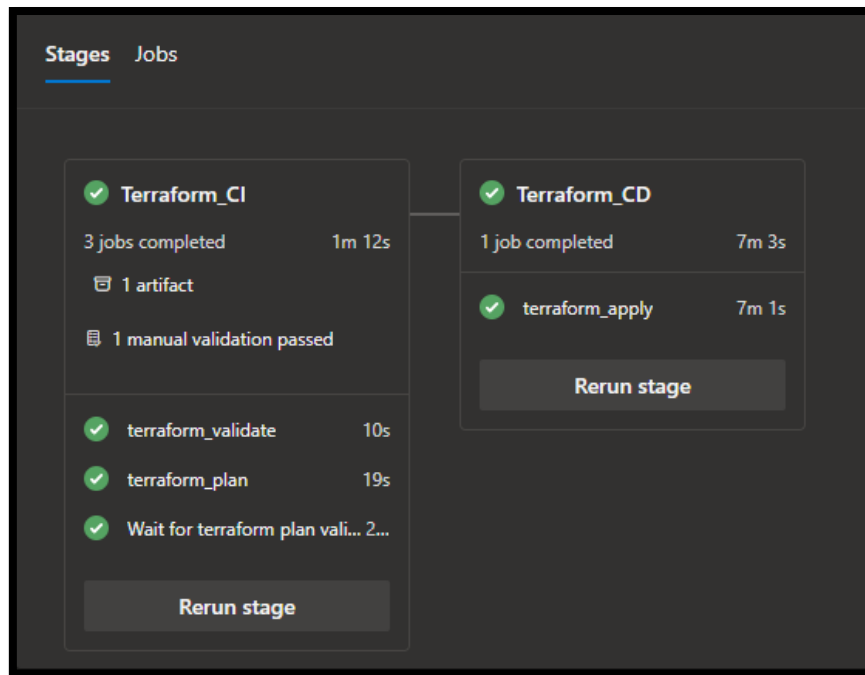


Figure 8 : Interface des pipelines Azure DevOps exécutés pour le déploiement du code Terraform

## 2.4) Construction des images Docker

Plongeons maintenant dans le processus de construction de nos agents Azure DevOps. Lorsque nous utilisons un agent Azure DevOps, ou un runner GitLab de manière similaire, au sein d'un cluster Kubernetes, il est essentiel de les encapsuler dans des conteneurs Docker. Pour rappel, La conteneurisation est une technologie qui permet d'emballer une application et ses dépendances dans un conteneur isolé. Un conteneur est une unité légère et autonome qui contient tout ce dont une application a besoin pour s'exécuter, y compris le code, les bibliothèques, les dépendances et les fichiers de configuration. Ces conteneurs offrent une isolation complète de l'environnement d'exécution, ce qui permet d'exécuter une application de manière cohérente et prévisible, quel que soit l'endroit où elle est déployée. Les conteneurs Docker offrent donc de nombreux avantages, tels que la portabilité, l'isolation et la répétabilité, ce qui en fait un choix idéal pour l'exécution d'applications dans des environnements Kubernetes.

Dans cette section, nous explorerons en détail ce qu'est une image basée sur un agent Azure DevOps, en mettant en évidence ses spécificités et les recommandations de Microsoft pour la construction de ces agents. Nous examinerons également toutes les étapes impliquées dans la configuration de nos agents personnalisés pour notre projet, ainsi que leur déploiement dans le registre de conteneurs.

La construction des images Docker de nos agents Azure DevOps est d'une grande importance, car elle détermine la manière dont ces agents seront exécutés dans notre environnement Kubernetes. Nous allons voir comment j'ai suivi les bonnes pratiques et les

recommandations de Microsoft pour garantir des performances optimales et une intégration harmonieuse avec l'infrastructure existante. Nous détaillerons chaque étape du processus de construction des images Docker, en soulignant les considérations clés, les configurations spécifiques et les bonnes pratiques à suivre. On va donc voir comment j'ai été en mesure de créer des images Docker robustes et adaptées aux besoins spécifiques du projet.

### 2.4.1) Déploiement d'un Self-Hosted-Agent

Avant d'entrer dans les détails du déploiement d'un agent autohébergé (Self-Hosted-Agent) dans Azure DevOps, il est important de comprendre ce qu'est réellement un agent autohébergé et de rappeler la différence entre cet agent et l'agent par défaut fourni par Azure DevOps. Un agent autohébergé est un agent Azure DevOps qui s'exécute sur notre propre infrastructure, que ce soit dans votre réseau local ou dans un environnement cloud privé. Contrairement à l'agent par défaut fourni par Azure DevOps, qui s'exécute sur les machines virtuelles Azure gérées par Microsoft, l'agent autohébergé vous permet d'avoir un contrôle total sur l'infrastructure sur laquelle l'agent est exécuté. Cela peut être particulièrement utile dans les scénarios où vous avez des exigences spécifiques en matière de sécurité, de connectivité réseau ou de configuration système.

Azure offre la possibilité d'avoir des agents autohébergés basés sur différents systèmes d'exploitation, ici Linux et Windows. Pour les agents Linux, les images sont basées sur Ubuntu 18.04 ou 20.04, qui sont des images officielles disponibles sur Docker Hub et qui sont très utilisées dans la communauté des conteneurs. Pour les agents autohébergés basés sur Windows, ils s'appuient sur une image Windows Server 2019 construite par Microsoft, qui est hébergée sur leur propre registre appelé "Microsoft Artifact Registry". Cependant, étant donné que nos besoins sont plus adaptés à l'environnement Linux, nous nous concentrerons sur les agents Ubuntu pour notre déploiement (voir [l'annexe](#) pour Dockerfile Windows).

```
FROM ubuntu:20.04
RUN DEBIAN_FRONTEND=noninteractive apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get upgrade -y

RUN DEBIAN_FRONTEND=noninteractive apt-get install -y -qq --no-install-recommends \
    apt-transport-https \
    apt-utils \
    ca-certificates \
    curl \
    git \
    iputils-ping \
    jq \
    lsb-release \
    software-properties-common

RUN curl -sL https://aka.ms/InstallAzureCLIDeb | bash

# Can be 'linux-x64', 'linux-arm64', 'linux-arm', 'rhel.6-x64'.
ENV TARGETARCH=linux-x64

WORKDIR /azp

COPY ./start.sh .
RUN chmod +x start.sh

ENTRYPOINT [ "./start.sh" ]
```

Figure 9 : Dockerfile par défaut pour le build d'un agent Azure DevOps sur Ubuntu 20:04



Commençons par examiner le Dockerfile fourni par Microsoft. Ce Dockerfile présente la configuration par défaut de l'agent Azure DevOps basé sur Ubuntu 20.04. Les instructions fournies permettent d'installer l'agent et les packages requis, avant de passer à la personnalisation de l'image en fonction de nos besoins.

Ce Dockerfile commence par définir l'image de base **ubuntu:20.04**. Ensuite, il met à jour le système d'exploitation Ubuntu et installe les mises à jour disponibles. Les packages requis, tels que **apt-transport-https**, **apt-utils**, **ca-certificates**, **curl**, **git**, **iputils-ping**, **jq**, **lsb-release** et **software-properties-common**, mais aussi Azure CLI, sont également installés. Le script **start.sh** est copié dans le répertoire de travail **/azp** et les droits d'exécution lui sont accordés. Enfin, l'instruction **ENTRYPOINT** spécifie que le script **start.sh** doit être exécuté lorsque le conteneur démarre.

```
#!/bin/bash
set -e

if [ -z "$AZP_URL" ]; then
    echo 1>&2 "error: missing AZP_URL environment variable"
    exit 1
fi

if [ -z "$AZP_TOKEN_FILE" ]; then
    if [ -z "$AZP_TOKEN" ]; then
        echo 1>&2 "error: missing AZP_TOKEN environment variable"
        exit 1
    fi
    AZP_TOKEN_FILE=/azp/.token
    echo -n $AZP_TOKEN >"$AZP_TOKEN_FILE"
fi

unset AZP_TOKEN

if [ -n "$AZP_WORK" ]; then
    mkdir -p "$AZP_WORK"
fi

export AGENT_ALLOW_RUNASROOT="1"

cleanup() {
    if [ -e config.sh ]; then
        print_header "Cleanup. Removing Azure Pipelines agent..."

        # If the agent has some running jobs, the configuration removal process will fail.
        # So, give it some time to finish the job.
        while true; do
            ./config.sh remove --unattended --auth PAT --token $(cat "$AZP_TOKEN_FILE") && break
            echo "Retrying in 30 seconds..."
            sleep 30
        done
    fi
}

print_header() {
    lightcyan='\033[1;36m'
    nocolor='\033[0m'
    echo -e "${lightcyan}$1${nocolor}"
}

# Let the agent ignore the token env variables
export VSO_AGENT_IGNORE=AZP_TOKEN,AZP_TOKEN_FILE

print_header "1. Determining matching Azure Pipelines agent..."

AZP_AGENT_PACKAGES=$(curl -Ls \
    -u user:$(cat "$AZP_TOKEN_FILE") \
    -H 'Accept:application/json;' \
    "$AZP_URL/_apis/distributedtask/packages/agent?platform=$TARGETARCH&top=1")

AZP_AGENT_PACKAGE_LATEST_URL=$(echo "$AZP_AGENT_PACKAGES" | jq -r '.value[0].downloadUrl')
```

Figure 10 : Extrait du script de démarrage du conteneur de l'agent Ubuntu

Ce script joue un rôle central dans la configuration et l'exécution de l'agent Azure DevOps sur Linux (voir [l'annexe](#) pour script PowerShell pour Windows). Il s'acquitte de plusieurs tâches essentielles, notamment la vérification des variables d'environnement **AZP\_URL** et **AZP\_TOKEN**, qui doivent être définies avec l'URL de l'instance Azure DevOps et le jeton d'authentification respectivement. Si ces variables ne sont pas définies, une erreur est signalée. Si la variable d'environnement **AZP\_WORK** est définie, un répertoire approprié est créé pour stocker le travail de l'agent. Les fonctions **cleanup** et **print\_header** sont utilisées respectivement pour supprimer la configuration de l'agent existant, si nécessaire, et pour afficher des messages d'en-tête. Ensuite, le script interroge l'URL de l'instance Azure DevOps pour déterminer l'agent Azure Pipelines correspondant à l'architecture cible spécifiée. Après avoir téléchargé et extrait l'agent Azure Pipelines à partir de l'URL déterminée, la configuration de l'agent est effectuée à l'aide du fichier **config.sh** et des paramètres appropriés tels que le nom de l'agent, l'URL de l'instance Azure DevOps, le jeton d'authentification, le pool et le répertoire de travail. Enfin, l'agent Azure Pipelines est exécuté à l'aide du script **run.sh**, permettant ainsi d'exécuter les travaux qui lui sont assignés.

#### 2.4.2) Personnalisation des images

Dans le cadre de l'intégration de notre premier projet au sein de notre solution de cluster privé, nous avons dû prendre en compte les besoins spécifiques de cet environnement. Pour cela, des études et des réunions ont été organisées avec l'ingénieur cloud responsable de l'écriture du code Terraform pour le projet. Ces interactions nous ont permis de comprendre le contexte du projet, les packages prérequis, ainsi que les versions spécifiques des applications requises pour le déploiement de l'infrastructure. À partir de ces informations, j'ai rédigé un Dockerfile personnalisé répondant aux besoins du déploiement de l'infrastructure tout en appliquant les bonnes pratiques pour la construction d'une image Docker optimisée.

```
FROM ubuntu:20.04

# Update the package repository and upgrade existing packages
RUN DEBIAN_FRONTEND=noninteractive apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get upgrade -y

# Install essential packages
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y -qq --no-install-recommends \
    apt-transport-https \
    apt-utils \
    ca-certificates \
    curl \
    git \
    iputils-ping \
    jq \
    lsb-release \
    software-properties-common

# Install Azure CLI
RUN curl -sL https://aka.ms/InstallAzureCLIDeb | bash

# Set the target architecture (Can be 'linux-x64', 'linux-arm64', 'linux-arm', 'rhel.6-x64')
ENV TARGETARCH=linux-x64

# Set the working directory
WORKDIR /azp

# Install Terraform & prerequisite packages
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y -qq --no-install-recommends unzip
# Terraform 1.2.0
RUN curl -LO https://releases.hashicorp.com/terraform/1.2.0/terraform_1.2.0_linux_amd64.zip && \
    unzip terraform_1.2.0_linux_amd64.zip && \
    mv terraform /usr/local/bin/

# Cleanup
RUN rm terraform_1.2.0_linux_amd64.zip && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# Copy the start script
COPY ./start.sh .
RUN chmod +x start.sh

# Set the entrypoint
ENTRYPOINT [ "./start.sh" ]
```

Installation de Unzip et Terraform

Nettoyage de l'image

Figure 11 : Dockerfile optimisée pour le build d'un agent de déploiement d'infrastructure

Le code Terraform utilisé dans ce projet nécessitait la version 1.0.2 de l'outil pour fonctionner correctement. Il est essentiel de conserver cette version spécifique tant que le code n'a pas été mis à jour pour être compatible avec d'autres versions de Terraform. Pour répondre à cette exigence, nous avons ajouté les instructions appropriées au Dockerfile pour installer Terraform 1.2.0. De plus, nous avons inclus l'installation du package "unzip", requis pour extraire les fichiers nécessaires à l'installation de Terraform.

Nous avons également tenu compte de l'importance de garder nos images Docker aussi légères que possible. Un conteneur léger permet une meilleure performance, une diminution des temps de démarrage et une utilisation plus efficace des ressources. C'est pourquoi nous avons inclus l'instruction "RUN **cleanup**" dans le Dockerfile. Cette étape permet de supprimer les fichiers temporaires et inutiles après l'installation de Terraform et "unzip", garantissant ainsi une image Docker plus légère et mieux optimisée. La légèreté des images est cruciale dans le contexte de Kubernetes, car cela impacte directement la scalabilité et les performances de notre cluster privé.

Pour le déploiement de l'application, nous avons réalisé une étape similaire. Des échanges avec l'architecte applicatif nous ont permis d'identifier que l'application utilise principalement le package Node.js. En conséquence, nous avons créé une nouvelle image Docker spécifique permettant de déployer un agent pour l'application Node.js.

```
# Install Node.js
RUN curl -sL https://deb.nodesource.com/setup_14.x | bash - && \
    DEBIAN_FRONTEND=noninteractive apt-get install -y -qq --no-install-recommends nodejs
```

Figure 12 : Dockerfile optimisée pour le build d'un agent de déploiement applicatif

Le Dockerfile de cette nouvelle image comprend l'installation de Node.js en utilisant la version 14.x via le gestionnaire de paquets de Debian. L'utilisation de cette version spécifique est essentielle pour s'assurer que l'application est déployée avec les bonnes dépendances et les fonctionnalités requises. L'instruction « **curl -sL https://deb.nodesource.com/setup\_14.x | bash -** » permet de configurer le dépôt Node.js pour Debian, tandis que « **DEBIAN\_FRONTEND=noninteractive apt-get install -y -qq --no-install-recommends nodejs** » installe Node.js de manière non interactive, évitant ainsi tout besoin de confirmation utilisateur pendant l'installation. Grâce à cette approche, nous avons pu créer une image Docker optimisée pour le déploiement de l'application Node.js. Cela nous permet de garantir que l'agent d'exécution pour l'application est prêt à être déployé dans notre cluster Kubernetes privé, et qu'il fonctionne de manière fiable et efficace lors de l'exécution de nos pipelines CI/CD.

La dernière étape du processus consiste à pousser l'image personnalisée dans le registre de conteneurs Azure. Comme rappelé précédemment, notre registre de conteneurs Azure est privé et dispose d'un accès restreint, ce qui garantit la sécurité de nos images. Pour ce faire, nous avons utilisé une machine virtuelle de gestion dans le même réseau virtuel que l'Azure Container Registry et l'Azure Kubernetes Service pour déployer les images dans le registre de conteneurs. Nous avons également utilisé un compte de service Azure ayant les droits "AcrPush" pour effectuer cette opération. Il est essentiel de suivre une nomenclature spécifique pour les noms et les tags des images afin de les reconnaître facilement parmi les autres. Voici les commandes permettant de pousser l'image dans le registre de conteneurs Azure :

Connexion à Azure	<code>az login --service-principal --username &lt;sp_id&gt; --password &lt;passwd&gt; --tenant &lt;azure_tenant_id&gt;</code>
Connexion à l'ACR	<code>docker login &lt;acr_name&gt;.azurecr.io</code>
Tag de l'image	<code>docker tag &lt;image_name&gt; &lt;acr_name&gt;.azurecr.io/runner-&lt;project_name&gt;-&lt;env&gt;:&lt;tag&gt;</code>
Push de l'image	<code>docker push &lt;acr_name&gt;.azurecr.io/runner-&lt;project_name&gt;-&lt;env&gt;:&lt;tag&gt;</code>

Toutes ces étapes décrites précédemment doivent être réalisées lors de chaque nouveau projet dont le déploiement se fera via ce nouveau cluster Kubernetes privé. Ce projet implique donc d'avoir un document d'exploitation visant à décrire toutes les étapes à réaliser au début d'un projet de déploiement d'application dans Azure.

## 2.5) Déploiement dans Kubernetes

Après avoir déployé l'infrastructure cible et les images des agents de déploiement, nous entrons maintenant dans l'étape cruciale du déploiement dans Kubernetes. Cette partie représente le cœur technique du projet, car le cluster Kubernetes mis en place joue un rôle central dans la gestion de notre solution. Pour assurer une présentation claire et structurée, j'ai divisé l'explication du processus de déploiement dans Kubernetes en plusieurs étapes. Tout d'abord, je vais commencer par aborder les prérequis nécessaires pour le déploiement dans Kubernetes pour garantir un fonctionnement de notre solution. Ensuite, nous explorerons le déploiement initial des agents Azure DevOps dans le cluster, en mettant l'accent sur les configurations spécifiques nécessaires pour que les agents fonctionnent correctement. Nous entrerons ensuite dans les détails de l'intégration de Keda dans notre solution permettant la mise à l'échelle des pipelines. Enfin, je vais montrer la mise en place de templates de déploiement avec Helm. Ces templates nous permettront de simplifier le déploiement des agents Azure DevOps, en facilitant le processus de configuration et en améliorant l'efficacité du déploiement.

### 2.5.1) Prérequis

Avant de commencer à travailler sur notre cluster Kubernetes, certains prérequis doivent être pris en compte pour garantir un déploiement réussi. Tout d'abord, toutes les manipulations du cluster doivent être effectuées via la machine virtuelle d'opérations, car elle est la seule machine dans le réseau privé du cluster, assurant ainsi la sécurité et la gestion appropriée des ressources.

L'un des premiers prérequis importants est de lier notre ressource Azure Kubernetes Service à l'Azure Container Registry (ACR). Comme cela a été configuré dans le code Terraform, notre AKS et notre ACR sont déjà liés en termes de réseaux, car ils se trouvent dans deux sous-réseaux du même VNet. De plus, l'AKS dispose d'un rôle Azure AD lui permettant de récupérer les images Docker stockées dans le registre de conteneur. Cependant, pour que cela fonctionne correctement, nous devons « attacher » l'ACR au cluster AKS. Cela se fait à l'aide de la commande Azure CLI « `az aks update --attach-acr` », qui utilise les autorisations de l'utilisateur exécutant la commande pour créer l'attribution de rôle ACR. Ce rôle est attribué à l'identité managée kubelet, qui est responsable de l'exécution des tâches des pods dans le cluster. Une fois cette étape effectuée, le cluster est prêt à récupérer les images Docker depuis l'Azure Container Registry.

Ensuite, nous devons dédier un namespace pour notre travail dans le cluster. Pour rappel, un namespace Kubernetes est un moyen de diviser les ressources du cluster en groupes logiques, offrant une isolation et une organisation distinctes. Dans mon cas, j'ai créé un namespace nommé « devopsagent » avec la commande « `k create ns devopsagent` ». Pour faciliter la configuration sur notre machine OPS, j'ai également configuré kubectl pour pointer directement vers ce namespace avec la commande « `k config set-context --current --namespace=devopsagent` ».

Enfin, un dernier prérequis important concerne la configuration d'un agent pool dans Azure DevOps pour accueillir les agents déployés avec Kubernetes. Il est nécessaire de créer ce pool d'agent avant de commencer à déployer les pods d'agent dans le cluster, car ces derniers requièrent les URL et noms des pools d'agent Azure DevOps pour être correctement déployés. J'ai donc créé un simple pool d'agent que nous appellerons ici « pool1 » pour des raisons de confidentialité. Selon les projets intégrant notre solution de cluster Kubernetes privé, d'autres pools d'agent peuvent être créés ultérieurement.

### 2.5.2) Déploiement des Pods

Dans notre cluster Kubernetes, les agents Azure DevOps sont représentés par des pods. Pour rappel, un pod est une unité de base dans Kubernetes, qui peut contenir un ou plusieurs conteneurs. Dans notre cas, chaque pod représente un agent Azure DevOps et contient un conteneur configuré avec toutes les dépendances nécessaires pour exécuter les pipelines. Pour déployer rapidement ces pods avec les configurations nécessaires et de manière scriptée, nous utilisons des objets de type "Déploiement" (Deployment en anglais). Un déploiement dans Kubernetes est une ressource qui permet de spécifier l'état souhaité d'un ensemble de pods et de gérer leur déploiement de manière déclarative. Il définit le nombre de répliques (replicas) de pods souhaité, le nom du pod, l'image du conteneur à utiliser, et d'autres configurations liées au déploiement.

Dans notre cas, le déploiement d'un agent est divisé en deux catégories : les metadata et les specs. Les metadata contiennent des informations sur l'objet de déploiement telles que son nom, le namespace dans lequel l'agent est déployé, et le nom de l'application pour faciliter l'organisation des workloads dans notre cluster. La partie "specs", quant à elle, représente le cœur de la configuration des pods. On y spécifie le nombre de répliques souhaité (ici un seul car nous aurons de l'autoscaling avec Keda), le nom du pod, et l'image provenant de notre Azure Container Registry.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: projet-infra-agent-deployment
  namespace: devopsagent
  labels:
    app: projet-agent
spec:
  replicas: 1
  selector:
    matchLabels:
      app: projet-agent
  template:
    metadata:
      labels:
        app: projet-agent
    spec:
      containers:
        - name: projet-infra-agent
          image: acurname.azurecr.io/runner-projet-infra:terraform_1.2.0
          env:
            - name: AZP_URL
              valueFrom:
                secretKeyRef:
                  name: secret
                  key: AZP_URL
            - name: AZP_TOKEN
              valueFrom:
                secretKeyRef:
                  name: secret
                  key: AZP_TOKEN
            - name: AZP_POOL
              valueFrom:
                secretKeyRef:
                  name: secret
                  key: AZP_POOL_INFRA
          volumeMounts:
            - mountPath: /var/run/docker.sock
              name: docker-volume
      volumes:
        - name: docker-volume
          hostPath:
            path: /var/run/docker.sock
```

Metadata

Specs

Figure 13 : Fichier YAML de l'objet de type « deployment » pour la création d'un agent Azure DevOps

La particularité des images des agents Azure DevOps est qu'elles requièrent des variables d'environnement pour fonctionner correctement. Ces variables d'environnement sont utilisées dans le script d'entrypoint de l'image, comme expliqué précédemment. Il est donc essentiel que Kubernetes puisse posséder ces variables d'environnement et les fournir aux pods lors du déploiement. C'est là que les secrets Kubernetes entrent en jeu. Un secret Kubernetes est une ressource qui permet de stocker des données sensibles, telles que des informations d'authentification ou des clés d'accès, de manière sécurisée. Dans notre cas, nous utilisons un objet secret (voir [annexe/Kubernetes](#)) pour stocker trois données : AZP\_URL pour l'URL de l'organisation Azure DevOps, AZP\_POOL pour le nom de l'agent pool (dans notre cas, "pool1"), et AZP\_TOKEN, qui est le token Azure DevOps permettant de déployer des agents dans l'agent pool. Il est important de préciser qu'inclure des secrets en base64 directement dans le code Kubernetes n'est pas une pratique sécurisée, car ces valeurs peuvent être facilement retrouvées. Une amélioration future consisterait donc à faire en sorte que Kubernetes récupère ces secrets directement depuis notre Azure Key Vault.

Avec ce déploiement, nous avons maintenant un agent fonctionnel. On va à présent voir comment mettre à l'échelle ces agents avec Keda et comment j'ai développé des templates de déploiement avec Helm pour simplifier le déploiement des agents.

### 2.5.3) Mise à l'échelle avec KEDA

Le déploiement de l'infrastructure cible et des agents Azure DevOps constitue une étape cruciale de notre projet, mais ce n'est qu'une partie du processus. Une fois l'infrastructure en place, nous devons être prêts à exécuter nos pipelines Azure DevOps et à gérer le flux de travail efficacement. Pour atteindre cet objectif, nous avons intégré Keda (Kubernetes-based Event Driven Autoscaler) dans notre architecture pour faciliter la mise à l'échelle automatique des agents déployés selon les besoins des pipelines et exécuter des jobs en parallèle.

L'intérêt de l'autoscaling réside dans l'optimisation des ressources et l'amélioration des performances lors de l'exécution de plusieurs jobs de pipelines Azure DevOps simultanément. Plutôt que de maintenir un nombre fixe d'agents actifs en permanence, Keda ajuste automatiquement le nombre d'agents en fonction de la charge de travail actuelle. Ainsi, lorsque de nouveaux jobs sont lancés, Keda déploie des agents supplémentaires pour répondre à la demande, et inversement, il réduit le nombre d'agents lorsque la charge diminue. Cela permet d'optimiser l'utilisation des ressources et d'assurer une exécution efficace des jobs de pipelines, tout en évitant de maintenir des agents inactifs.

Keda est un autoscaler basé sur Kubernetes qui permet de gérer dynamiquement le nombre de réplicas (agents) de nos déploiements selon la charge des ressources. Il s'intègre facilement dans l'environnement Kubernetes existant et peut être configuré pour surveiller différentes sources de métriques pour déterminer quand et comment mettre à l'échelle les déploiements. Keda est basé sur le modèle de Custom Resource Definitions (CRD) de Kubernetes et utilise les métriques fournies par des adaptateurs pour déterminer le niveau de mise à l'échelle requis.

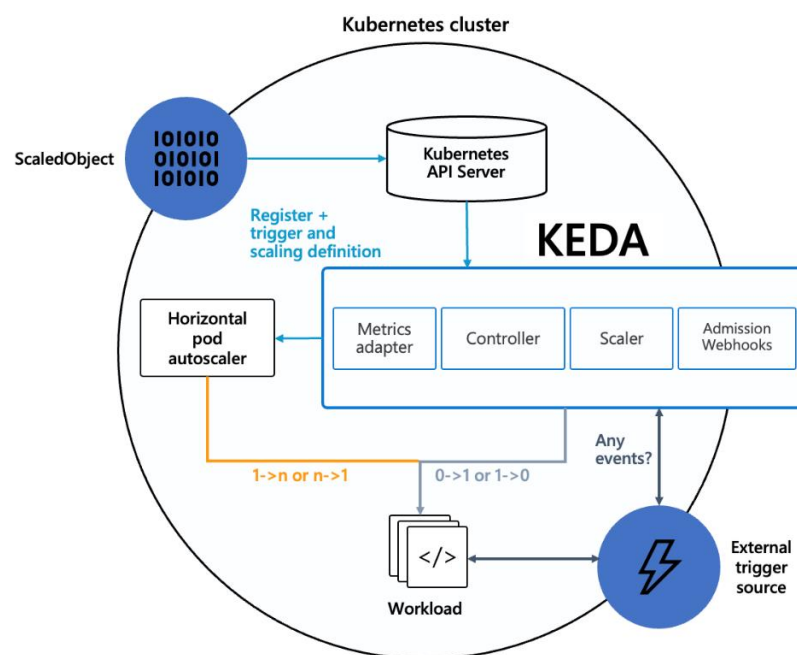


Figure 14 : Architecture de Keda



Keda est capable de faire le lien avec Azure Pipelines grâce à son adaptateur Azure Pipelines. Cet adaptateur est capable de surveiller l'état des builds et des releases dans Azure Pipelines et de récupérer des métriques pertinentes telles que le nombre de builds en attente, le temps d'exécution moyen, etc. En se basant sur ces métriques, Keda détermine quand il est nécessaire de mettre à l'échelle les agents pour répondre à la demande des pipelines.

```
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: pipeline-trigger-secret
  namespace: devopsagent
spec:
  secretTargetRef:
    - parameter: personalAccessToken
      name: secret ← Nom du secret de l'agent à scaler
      key: AZP_TOKEN
---
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: azure-pipelines-scaledobject
  namespace: devopsagent
spec:
  scaleTargetRef:
    name: projet-infra-agent-deployment ← Déploiement ciblé
  minReplicaCount: 1
  maxReplicaCount: 4 ← Nombre maximum d'agent à déployer
  cooldownPeriod: 3
  triggers:
    - type: azure-pipelines
      metadata:
        poolID: "1" ← ID du pool d'agent dans lequel est déployé l'agent à scaler
        organizationURLFromEnv: "AZP_URL"
      authenticationRef:
        name: pipeline-trigger-secret
```

Figure 15 : Fichier YAML pour la mise à l'échelle avec Keda d'un agent Azure DevOps

Dans notre configuration, nous avons créé un objet Keda « **ScaledObject** » qui définit les paramètres de mise à l'échelle pour nos agents Azure DevOps. Nous avons spécifié le déploiement cible et défini le nombre minimum de réplicas (**minReplicaCount**) à 1 pour garantir qu'il y a toujours au moins un agent actif pour exécuter les jobs de manière efficace. Le nombre maximum de réplicas (**maxReplicaCount**) a été fixé à 4 dans cet exemple pour limiter le nombre d'agents en cours d'exécution simultanément, mais le chiffre variera en fonction du projet.

De plus, nous avons configuré un « **cooldownPeriod** » de 3 secondes pour éviter de mettre à l'échelle de manière trop réactive et réduire les fluctuations indésirables. Ce YAML spécifie que Keda doit surveiller les métriques d'Azure Pipelines en utilisant l'adaptateur azure-pipelines. Pour sécuriser les connexions, nous avons également créé un objet « **TriggerAuthentication** » qui fait référence au secret contenant le token d'accès Azure DevOps nécessaire à l'adaptateur pour interagir avec les pipelines.

En utilisant cette configuration, Keda surveille continuellement les pipelines Azure DevOps, et lorsque de nouveaux jobs sont en attente, il met automatiquement à l'échelle le nombre d'agents pour répondre à la demande. Cette mise à l'échelle automatique permet d'optimiser l'utilisation des ressources et d'améliorer l'efficacité de l'exécution des pipelines Azure DevOps.

#### 2.5.4) Création de Helm Charts

Avec notre infrastructure de cluster Kubernetes opérationnelle, nous prévoyons de déployer de nombreux agents Azure DevOps, à la fois pour l'application et l'infrastructure de chaque projet. Dans cet environnement, il devient crucial de pouvoir redéployer facilement des agents avec seulement quelques modifications de configuration, sans avoir à tout recréer à partir de zéro. C'est là qu'intervient Helm, un outil qui nous permet de gérer les déploiements d'applications sur Kubernetes avec une approche infrastructure-as-code.

Helm est souvent considéré comme le "gestionnaire de paquets" de Kubernetes. Il nous permet de créer, partager et déployer des "Helm Charts", qui sont des packages préconfigurés contenant toutes les ressources et configurations nécessaires pour déployer une application sur Kubernetes. Les Helm Charts fournissent un moyen simple de définir et d'organiser des déploiements, des services, des secrets et d'autres objets Kubernetes dans des fichiers YAML, tout en autorisant la variabilisation, la réutilisation et l'automatisation.

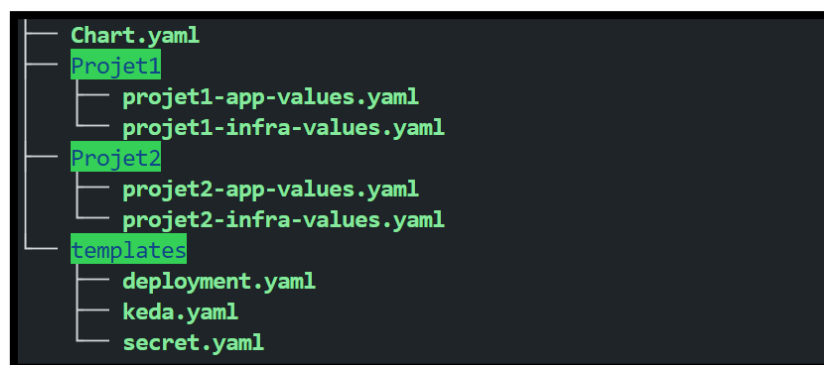


Figure 16 : Structure du Chart Helm pour le déploiement d'un agent

Le Helm Chart que j'ai créé pour déployer nos agents Azure DevOps est une représentation concrète de notre infrastructure-as-code. Il contient des fichiers de modèle (templates) qui définissent les ressources Kubernetes, tels que le déploiement de l'agent, le secret contenant les informations d'authentification Azure DevOps, ainsi que le fichier de configuration Keda pour la mise à l'échelle automatique des agents, en somme, tous les fichiers vu précédemment. Les valeurs dans les fichiers de valeurs (values) nous permettent de personnaliser chaque déploiement d'agent avec des paramètres spécifiques, tout en gardant la structure générale du Helm Chart.

L'intérêt principal d'utiliser Helm est la variabilisation. Grâce aux fichiers de valeurs, nous pouvons facilement modifier les paramètres pour chaque projet, sans avoir besoin de

toucher aux modèles YAML sous-jacents. Cela facilite grandement la maintenance de nos déploiements, en nous permettant de garder une configuration claire et concise pour chaque application et infrastructure de projet. De plus, Helm nous permet de réutiliser les Helm Charts dans d'autres projets, ce qui simplifie le processus de déploiement pour des cas d'utilisation similaires.

```
labels:
  app: projet1

image:
  repository: acrname.azurecr.io/projet1-siteassist-app
  tag: ubuntu_20.04

data:
  azp_pool: XXXXXXXXXX

maxReplicaCount: 4
poolID: "1"
```

Figure 17 : Exemple d'un fichier de values pour le déploiement d'un agent pour les pipelines applicatif

D'un point de vue technique, la création des Helm Charts pour nos agents Azure DevOps est relativement simple. Pour chaque nouveau projet, nous créons un nouveau dossier contenant les fichiers de modèle pour l'application et l'infrastructure (s'il y a lieu), ainsi que les fichiers de valeurs correspondants. Nous personnalisons ensuite les valeurs spécifiques à chaque projet dans les fichiers de valeurs. Ainsi, lorsqu'il est temps de déployer un nouvel agent pour un projet donné, nous n'avons qu'à exécuter la commande Helm « **helm install** » avec les valeurs appropriées pour ce projet, et l'agent est déployé automatiquement selon les spécifications définies dans le Helm Chart.

Cette approche modulaire et automatisée nous permet de gagner du temps et de réduire les risques d'erreurs lors du déploiement de nouveaux agents dans notre cluster Kubernetes. De plus, elle garantit une cohérence et une reproductibilité élevées dans l'ensemble de notre infrastructure, tout en facilitant l'ajout de nouvelles fonctionnalités à mesure que nos besoins évoluent. Avec Helm, nous sommes mieux armés pour gérer efficacement notre infrastructure Kubernetes et répondre aux demandes changeantes de déploiement d'agents Azure DevOps dans notre environnement.

### III. Bilan

#### 3.1) Résultats obtenus

La version 1 de ce projet s'est révélée être une réussite à tous égards. Chacune des phases de test a été menée avec succès pour garantir un déploiement robuste et fiable de notre solution. Initialement, j'avais réalisé des tests en déployant les agents en tant que conteneurs en dehors d'Azure Kubernetes Service (AKS) afin de nous assurer de leur bon fonctionnement et de leur intégration avec Azure DevOps. Suite à cela, j'ai procédé au déploiement des agents en utilisant des Helm Charts, ce qui nous a permis de confirmer que les templates étaient fonctionnels et prêts à être utilisés dans d'autres projets. La mise en œuvre de Keda pour le scaling automatique des agents selon les pipelines s'est aussi avérée très concluante. En effet, j'ai créé un pipeline de test (voir [annexe/Tests](#)) permettant de réaliser plusieurs jobs en parallèle en affichant pour chacun le nom de l'agent sur lequel le job est exécuté. Ces tests ont montré que des agents supplémentaires sont automatiquement créés pour exécuter plusieurs jobs de pipelines en parallèle, ce qui permet une exécution plus rapide des pipelines et une utilisation plus efficace des ressources.

Name	Last run	Current status	Agent version
agent-775dcd7df-brbcw	Now	Running build 20230725.1	3.220.5
agent-775dcd7df-mq47d		Idle	3.220.5
agent-775dcd7df-pb7xm		Idle	3.220.5
agent-775dcd7df-tsphv		Idle	3.220.5

Figure 18 : Pool d'agent utilisé pour l'exécution de mes pipelines de tests

Mes collègues d'AMCS, du côté applicatif comme du côté infrastructure, ont exprimé leur satisfaction concernant cette version 1 de la solution. Ils ont constaté un gain de temps significatif lors de l'exécution des pipelines et une meilleure efficacité globale grâce à l'utilisation des agents personnalisés, contrairement aux agents par défaut de Microsoft. Cependant, Il est important de souligner que cette version est encore à l'état de V1, et il reste certains points à améliorer.

En cours de développement, je travaille sur l'isolation de l'Azure Container Registry (ACR) en le plaçant dans un private endpoint pour renforcer la sécurité. De plus, comme montrer précédemment, il est prévu que le cluster Kubernetes récupère les secrets Azure DevOps à partir d'Azure Key Vault pour renforcer la gestion des secrets. L'un des aspects clés en cours de développement est le peering entre les différents projets. Pour le moment, les agents permettent de déployer des applications et des infrastructures non privées. Cependant, ils ne peuvent pas encore déployer dans des environnements privés (notamment dans des bases de données liées à des private endpoint). Nous prévoyons de remédier à cette limitation bientôt en collaboration avec l'ingénieur cloud afin de permettre un déploiement complet dans tous les environnements.

En conclusion, cette première version du projet a été un succès, avec des résultats positifs et des retours enthousiastes de l'équipe AMCS. Je suis confiant que les améliorations en cours apporteront des fonctionnalités supplémentaires et une utilisation encore plus performante de la solution dans le futur.

### 3.2) Analyse et avis sur le sujet

Avec mon analyse du sujet, je veux mettre en évidence des points importants concernant l'utilisation d'Azure DevOps pour le déploiement applicatif. Globalement, l'utilisation d'Azure DevOps s'est avérée très intéressante grâce à sa liaison étroite avec Azure, permettant une centralisation des technologies pour tous les projets Azure. Cela facilite l'utilisation des mêmes services, les mêmes principaux de service, et offre une expérience homogène pour les équipes travaillant sur des projets Azure.

Cependant, l'un des points critiques que j'ai relevés concerne le principe tarifaire d'Azure DevOps. Le fait de devoir payer pour exécuter des jobs en parallèle, qui constitue pourtant le cœur de notre projet, peut s'avérer limitant. Avec un coût de 15€ par mois par agent exécuté en parallèle, il est nécessaire de prévoir à l'avance le nombre de jobs qui seront exécutés en parallèle. Cette limitation peut poser des problèmes lorsqu'il y aura de nombreux projets nécessitant des déploiements en parallèle. Dans de tels cas, il pourrait être nécessaire d'explorer d'autres solutions de déploiement pour éviter de dépasser les limites budgétaires.

Du point de vue des pipelines, des runners et de l'expérience globale de développement, j'ai personnellement préféré l'expérience de pipeline sur GitLab. Notamment, sur les environnements Azure et AWS, GitLab permet une approche plus commune pour déployer entre notre équipe travaillant sur Azure et celle travaillant sur AWS. En comparaison, bien qu'Azure DevOps fonctionne pour tous les fournisseurs de services cloud, il est principalement conçu pour les environnements Azure. J'ai également hâte de comparer cette expérience avec GitHub, même si je n'ai pas encore eu l'occasion de travailler avec cette plateforme.

En conclusion, L'utilisation d'Azure DevOps pour le déploiement a été très intéressante en raison de son intégration avec Azure, mais le modèle tarifaire peut être limitant pour les déploiements en parallèle. Globalement, ce projet a ouvert de nouvelles opportunités d'amélioration pour l'avenir tout en démontrant une amélioration significative dans nos processus de déploiement.

### 3.3) Apport personnel et apprentissage

La réalisation de ce projet a été une expérience totalement personnelle, car mis à part les phases d'architecture où j'ai reçu de l'aide de l'architecte, j'ai géré l'ensemble du déploiement en solo. Cette opportunité m'a permis de monter en compétence de manière considérable. Bien que je possédais déjà une bonne connaissance de Docker et Terraform, ce projet m'a permis de me perfectionner dans ces domaines. En revanche, je n'avais pas une grande expérience avec Kubernetes, mais grâce à ce projet, j'ai considérablement amélioré mes compétences dans son utilisation, ainsi que dans l'utilisation de Helm. Aujourd'hui, je me sens pleinement opérationnel sur ces sujets, ce qui représente une réelle valeur ajoutée pour moi-même et pour l'équipe.

Le projet a apporté une contribution significative à l'équipe, et je peux directement voir l'impact positif de sa finalisation. En aidant mes collègues à améliorer leurs processus de déploiement, ce sujet a grandement facilité leur travail quotidien. Je constate une augmentation de l'efficacité dans les pipelines, ce qui se traduit par un gain de temps précieux pour l'équipe. Savoir que ce projet est utile à mes collègues me motive davantage à continuer à contribuer de manière positive et à explorer de nouvelles possibilités d'amélioration.

En ce qui concerne le concept du DevOps, j'avais déjà une certaine compréhension théorique, mais il restait quelque chose d'assez abstrait pour moi. Cependant, tout au long de ce projet, j'ai pu véritablement comprendre le fonctionnement du DevOps. En tant qu'intermédiaire entre les développeurs et les opérations, j'ai dû faciliter et automatiser leur travail quotidien. Cette collaboration étroite entre les différents acteurs humains du projet, en harmonie avec les outils et les technologies, représente véritablement l'essence du DevOps.

En conclusion, ce projet m'a apporté une montée en compétence considérable dans divers domaines, notamment Docker, Terraform, Kubernetes et Helm. Son utilité concrète au sein de l'équipe m'a donné une vision claire de l'impact positif que peut avoir une solution bien pensée et bien déployée. En outre, ce projet m'a permis de saisir pleinement le concept du DevOps, en devenant le maillon essentiel entre les développeurs et les opérations pour améliorer les processus et collaborer de manière harmonieuse. Au-delà des compétences techniques, ce projet m'a également appris l'importance de la collaboration et de l'efficacité au sein d'une équipe. Ainsi, je termine ce mémoire en réalisant que le DevOps n'est pas seulement une approche technique, mais une culture de travail qui prône la collaboration, l'automatisation et l'amélioration continue pour des déploiements réussis et des équipes épanouies.

## Annexes

### ➤ Docker

```
FROM mcr.microsoft.com/windows/servercore:ltsc2019

WORKDIR /azp

COPY start.ps1 .

CMD powershell .\start.ps1
```

Figure 19 : Dockerfile par défaut pour le build d'un agent Azure DevOps sur Windows Server 2019

```
if (-not (Test-Path Env:AZP_URL)) {
    Write-Error "error: missing AZP_URL environment variable"
    exit 1
}

if (-not (Test-Path Env:AZP_TOKEN_FILE)) {
    if (-not (Test-Path Env:AZP_TOKEN)) {
        Write-Error "error: missing AZP_TOKEN environment variable"
        exit 1
    }
    $Env:AZP_TOKEN_FILE = "\azp\token"
    $Env:AZP_TOKEN | Out-File -FilePath $Env:AZP_TOKEN_FILE
}

Remove-Item Env:AZP_TOKEN

if ((Test-Path Env:AZP_WORK) -and -not (Test-Path $Env:AZP_WORK)) {
    New-Item $Env:AZP_WORK -ItemType directory | Out-Null
}

New-Item "\azp\agent" -ItemType directory | Out-Null

# Let the agent ignore the token env variables
$Env:VSO_AGENT_IGNORE = "AZP_TOKEN,AZP_TOKEN_FILE"

Set-Location agent

Write-Host "1. Determining matching Azure Pipelines agent..." -ForegroundColor Cyan

$base64AuthInfo = [Convert]::ToBase64String([Text.Encoding]::ASCII.GetBytes(":${Get-Content $Env:AZP_TOKEN_FILE}"))
$package = Invoke-WebRequest -Headers @{Authorization = ("Basic $base64AuthInfo")} "${Env:AZP_URL}/_apis/distributedtask/packages/agent?platform=win-x64&$top=1"
$packageUrl = $package[0].Value.downloadUrl

Write-Host $packageUrl

Write-Host "2. Downloading and installing Azure Pipelines agent..." -ForegroundColor Cyan

$wc = New-Object System.Net.WebClient
$wc.DownloadFile($packageUrl, "$(Get-Location)\agent.zip")

Expand-Archive -Path "agent.zip" -DestinationPath "\azp\agent"

try {
    Write-Host "3. Configuring Azure Pipelines agent..." -ForegroundColor Cyan

    .\config.cmd --unattended `
        --agent "${if (Test-Path Env:AZP_AGENT_NAME) { $Env:AZP_AGENT_NAME } else { hostname }}" `
        --url "${if (Test-Path Env:AZP_URL) { $Env:AZP_URL }}" `
        --auth PAT `
        --token "${Get-Content $Env:AZP_TOKEN_FILE}" `
        --pool "${if (Test-Path Env:AZP_POOL) { $Env:AZP_POOL } else { 'Default' }}" `
        --work "${if (Test-Path Env:AZP_WORK) { $Env:AZP_WORK } else { '_work' }}" `
        --replace

    Write-Host "4. Running Azure Pipelines agent..." -ForegroundColor Cyan

    .\run.cmd
}
}
```

Figure 20 : Extrait du script PowerShell de démarrage du conteneur de l'agent Windows

## ➤ Terraform

```
#!/bin/sh

# Az CLI
curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash

# Docker
sudo apt update
sudo apt install apt-transport-https ca-certificates curl software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable"
sudo apt update
apt-cache policy docker-ce
sudo apt install docker-ce
sudo usermod -aG docker $USER
su - $(USER)

# Kubectl
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
echo 'source <(kubectl completion bash)' >> ~/.bashrc
echo 'alias k=kubectl' >> ~/.bashrc
echo 'complete -o default -F __start_kubectl k' >> ~/.bashrc
source ~/.bashrc

# Helm
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
chmod 700 get_helm.sh
./get_helm.sh
rm -f get_helm.sh
```

Figure 21 : Script shell inséré dans le customdata de la VM opération

## ➤ Kubernetes

```
apiVersion: v1
kind: Secret # Secrets for Azure DevOps authentication
metadata:
  name: secret
  namespace: devopsagent
data:
  AZP_URL: _____
  AZP_POOL: _____
  AZP_TOKEN: _____
```

Figure 22 : Fichier YAML pour la création des secrets d'un agent Azure DevOps

```
1 apiVersion: apps/v1
2 kind: Deployment # Deployment of the azure devops agent on Ubuntu containers
3 metadata:
4   name: {{ .Release.Name }}-agent
5   namespace: devopsagent
6   labels:
7     app: {{ .Values.labels.app }}-agent
8 spec:
9   replicas: 1
10  selector:
11    matchLabels:
12      app: {{ .Values.labels.app }}-agent
13  template:
14    metadata:
15      labels:
16        app: {{ .Values.labels.app }}-agent
17    spec:
18      containers:
19        - name: {{ .Release.Name }}-agent
20          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
21          env:
22            - name: AZP_URL
23              valueFrom:
24                secretKeyRef:
25                  name: {{ .Release.Name }}-pipeline-auth
26                  key: AZP_URL
27            - name: AZP_TOKEN
28              valueFrom:
29                secretKeyRef:
30                  name: {{ .Release.Name }}-pipeline-auth
31                  key: AZP_TOKEN
32            - name: AZP_POOL
33              valueFrom:
34                secretKeyRef:
35                  name: {{ .Release.Name }}-pipeline-auth
36                  key: AZP_POOL
37          volumeMounts:
38            - mountPath: /var/run/docker.sock
39              name: docker-volume
40          volumes:
41            - name: docker-volume
42              hostPath:
43                path: /var/run/docker.sock
```

Figure 23 : Template "deployment.yaml" pour le déploiement d'un Helm Chart



## ➤ Tests

```
trigger:
- none

stages:
  # Testing
  - stage: build

    pool:
      name: k8s-agent-pool

    jobs:
      - job: buildInParallel

        # Test Job in parallel
        strategy:
          matrix:
            build01:
              myVar: '1.0'
            build02:
              myVar: '1.0'
            build03:
              myVar: '1.0'
            build04:
              myVar: '1.0'
            build05:
              myVar: '1.0'
            build06:
              myVar: '1.0'
          maxParallel: 7

        steps:
          - script: |
              echo "Hello from inside Docker Container: $HOSTNAME"
              printenv
```

Figure 24 : Pipeline vérifiant le bon fonctionnement de la mise à l'échelle des agents

## Table des références

### Documentation :

Déploiement d'un Self-Hosted Agent sur Azure DevOps : <https://learn.microsoft.com/en-us/azure/devops/pipelines/agents/docker?view=azure-devops>

Documentation de KEDA : <https://keda.sh/docs/2.10/concepts/>

Mettre à l'échelle d'Azure Pipelines avec KEDA : <https://keda.sh/blog/2021-05-27-azure-pipelines-scaler/>

Déploiement d'une VM avec Terraform en utilisant des custom data : <https://medium.com/microsoftazure/custom-azure-vm-scale-sets-with-terraform-and-cloud-init-6a592dc41523>

Pricing Azure DevOps: <https://azure.microsoft.com/en-us/pricing/details/devops/azure-devops-services/>

### Vidéo :

Déploiement d'un Agent Azure DevOps avec Docker : <https://www.youtube.com/watch?v=KFykBO6fxlk>

Déploiement d'un Agent Azure DevOps dans Kubernetes : <https://www.youtube.com/watch?v=8yjBvwy9MT4&t=0s>

Mise à l'échelle des pipelines : [https://www.youtube.com/watch?v=G\\_3sl9XUxSq&t=0s](https://www.youtube.com/watch?v=G_3sl9XUxSq&t=0s)

Tutoriel d'un déploiement Kubernetes : <https://www.youtube.com/watch?v=HM8rLC107W0>

## Table des figures

Figure 1 : Fonctionnement de la CI/CD .....	6
Figure 2 : Organigramme de l'unité Cloud, Dataplatform & Apps de l'équipe AMCS.....	8
Figure 3 : Schéma d'architecture du projet.....	11
Figure 4 : Extrait du Terraform : Ressource Groupes et Réseaux Virtuel.....	18
Figure 5 : Extrait du Terraform : ACR et AKS .....	19
Figure 6 : Extrait du Terraform : VM d'opérations .....	20
Figure 7 : Extrait du pipeline Azure DevOps déployant le Terraform .....	21
Figure 8 : Interface des pipelines Azure DevOps exécutés pour le déploiement du code Terraform ....	23
Figure 9 : Dockerfile par défaut pour le build d'un agent Azure DevOps sur Ubuntu 20:04 .....	24
Figure 10 : Extrait du script de démarrage du conteneur de l'agent Ubuntu .....	25
Figure 11 : Dockerfile optimisée pour le build d'un agent de déploiement d'infrastructure .....	27
Figure 12 : Dockerfile optimisée pour le build d'un agent de déploiement applicatif.....	28
Figure 13 : Fichier YAML de l'objet de type « deployment » pour la création d'un agent Azure DevOps .....	31
Figure 14 : Architecture de Keda .....	32
Figure 15 : Fichier YAML pour la mise à l'échelle avec Keda d'un agent Azure DevOps.....	33
Figure 16 : Structure du Chart Helm pour le déploiement d'un agent .....	34
Figure 17 : Exemple d'un fichier de values pour le déploiement d'un agent pour les pipelines applicatif.....	35
Figure 18 : Pool d'agent utilisé pour l'exécution de mes pipelines de tests .....	36
Figure 19 : Dockerfile par défaut pour le build d'un agent Azure DevOps sur Windows Server 2019 .	39
Figure 20 : Extrait du script PowerShell de démarrage du conteneur de l'agent Windows .....	39
Figure 21 : Script shell inséré dans le customdata de la VM opération .....	40
Figure 22 : Fichier YAML pour la création des secrets d'un agent Azure DevOps .....	40
Figure 23 : Template "deployment.yaml" pour le déploiement d'un Helm Chart.....	40
Figure 24 : Pipeline vérifiant le bon fonctionnement de la mise à l'échelle des agents .....	41